Evaluation of Big Data Platforms for Industrial Process Data

Do Tuan Viet

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.7.2017

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

M.Sc. (Tech.) Olli Juhola

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Do Tuan Viet

Title: Evaluation of Big Data Platforms for Industrial Process Data

Date: 21.7.2017 Language: English Number of pages: 6+53

School of Science

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: M.Sc. (Tech.) Olli Juhola

When the number of IoT devices, as well as human activities on the Internet, has increased fast in recent years, data generated has also witnessed an exponential growth in volume. Therefore, various frameworks and software such as Cassandra, Hive, and Spark have been developed to store and explore this massive amount of data. In particular, the waves of Big Data have also reached the industrial businesses. As the number of sensors installed in machines and mills significantly increases, log data is generated from these devices in higher frequencies and enormously complex calculations are applied to this data. The thesis is aimed at evaluating how effectively the current Big Data frameworks and tools manipulate industrial Big Data, especially process data.

After surveying several techniques and potential frameworks and tools, the thesis focuses on building a prototype of a data pipeline. The prototype must satisfy a set of use cases. The data pipeline contains several components including Spark, Impala, and Sqoop. Also, it uses Parquet as the file format and stores the Parquet files in S3. Several experiments were also conducted in AWS, to validate the requirements in the use cases. The workload used for these tests was around 690 GBs of Parquet files. This amount of data includes one million channels, divided into one thousand groups, and the data sampling rate was one data point per second.

The results of the experiments show that the performance of current big data frameworks may fulfill the performance requirements and the features in the use cases and industrial businesses in general.

Keywords: Big Data, Hadoop, Spark SQL, Performance

I want to thank Associate Professor Keijo Heljanko, my instructor Olli Juhola and Karl Holmstrom for their good guidance.

Otaniemi, 21.7.2017

Tuan Viet Do

Contents

Abbreviations and Acronyms

HDFS	Hadoop Distributed File System
AWS	Amazon Web Service
S3	Simple Storage Service
EC2	Amazon Elastic Compute Cloud
RDD	Resilient Distributed Dataset
MPI	Message Passing Interface
GC	Garbage Collection
AST	Abstract Syntax Tree
MPP	Massive Parallel Processing
DDL	Data Definition Language
OLAP	Online Analytical Processing
DSL	Domain Specific Language
JVM	Java Virtual Machine
CDH	Cloudera Distribution Hadoop

1 Introduction

In recent years, the explosion of social networks, large-scale e-commerce websites such as Facebook and Amazon as well as billions of IoT devices connected to the Internet has resulted in an exponential growth of data to store and process. In a report [10], the amount of data is predicted to reach 40,000 Exabytes (1 Exabyte is equivalent to 1 billion Gigabytes) in 2020. Enterprises and companies around the world are looking for new solutions and techniques for updating their systems to tackle daily a massive and ever increasing amount of data. Savcor[1], providing tools for analyzing process data and other solutions for forestry information management, is also interested in solutions to handle a massive amount of data and how they perform. This thesis focuses on evaluating potential frameworks and tools on the big data world currently and building a prototype to examine its performance.

1.1 Process Industry and Big Data Trend

In this section, a data processing software in the process industry will be discussed through Wedge - a product of Savcor. The trend of big data in the industry will be described as well.

1.1.1 Description of the Wedge System

Wedge is a system for gathering process data, managing it and analyzing process phenomena. Wedge supports various data sources with different standards for connections to industrial devices and process control systems. Also, Wedge software provides users, such as production engineers and operators mathematical, tools to monitor and diagnose changes in the process, and then determine the root cause of process fluctuations. Wedge uses the client - server model. The components are:

- Wedge Server - A single server optionally including a columnar datastore (WDB) Savcor has developed on its own, and drivers connecting to various data sources using different standards such as OPC DA, HDA as well as JDBC/ODBC. Data will be pulled from external sources or third party databases for analysis. Data can be cached in WDB for the speed of queries.

- Wedge Client provides users a graphic interface to monitor the state of processes and industrial devices, mathematical tools to calculate and detect abnormal changes in numerous process signals.

The process of the whole system is illustrated in Figure 1. One of largest data sources which Wedge works with contains around 250,000 measurements also know as channels or tags, and the sampling interval of collecting data is one sample in 10 seconds. Therefore, the amount of data collected every day is around 10 GB per day. If this data is preserved up to 10 years, then the amount of data will be as much as 40 TB in total.

[1] https://www.savcor.com/

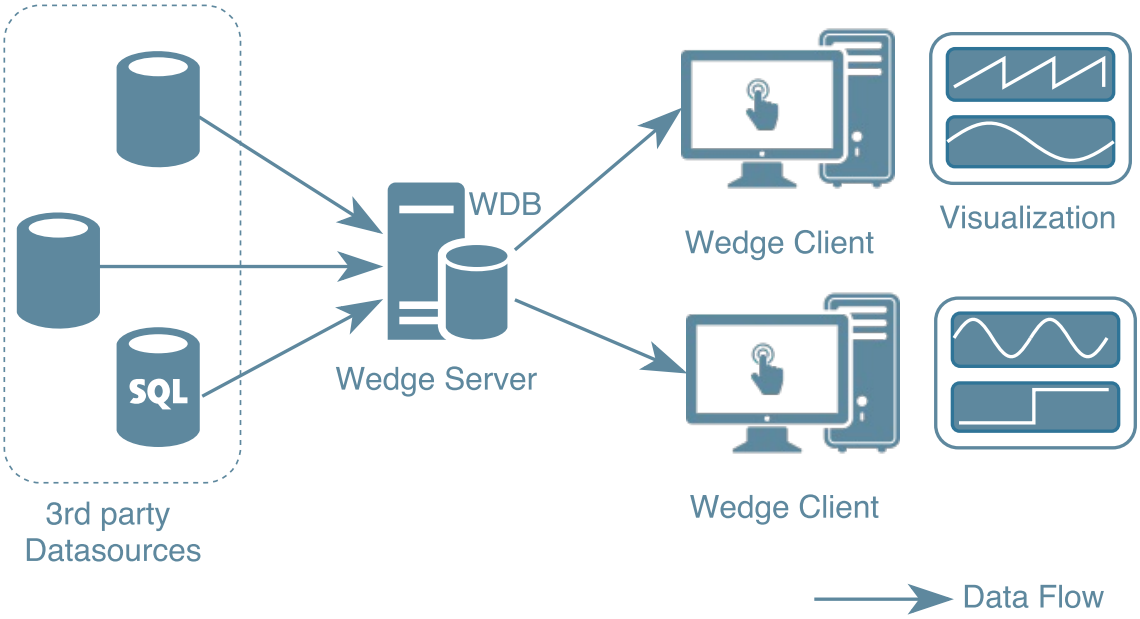

Figure 1: Wedge system and the data flow

1.1.2 Big Data Trend in the Process Industry

The fourth generation of industry [18] gradually emerges to the world when machines in factories and mills become connected. With the goal of automation, these machines exchange information, and based on this data to make decisions on their own without human involvement. Therefore, the amount of data to be transferred and processed is huge. Especially, in process industries, such as pulp and paper and metal, the amount of data becomes increasingly larger. The increase in the amount of data is caused by several reasons. Firstly, the number of sensors installed in the mills has increased significantly. Secondly, for monitoring a system more precisely, the sampling intervals to collect data will reduce. For example, from one data point per ten seconds previously to one data point per second. Thirdly, user requirements on complex aggregations involving a high number of measurements, so the amount of data will grow by orders of magnitude. For example, a system for storing and analyzing process data should store the data up to 0.5 PB which is equivalent to the amount of data in ten years from 250,000 simultaneous measurements with time interval being 1 sample per second.

1.2 The Goals of the Thesis

The prototyped system should have good performance in write operations to tackle high volume of input data as well as in read operations to maintain smooth interaction with users (at least 1,000,000 data points per second). Moreover, the system should be highly scalable, adding more computing resources on demand. The system can be deployed in customer's local system or quickly moved to a cloud infrastructure

later on. These requirements of the system will be translated into different use cases detailed in next section. All use cases are implemented in the current system and evaluated with a massive amount of measurements to benchmark performance.

1.2.1 Use Cases

There are several goals for the system, which are enumerated as below.

1. Feeding raw input data in real time:
   Raw input data may come from various data sources chronologically, and they have the same sample rate which is a sampling per second. However, the number of individual channels can reach one million. Therefore, the rate of incoming data is quite high at one million samples per second. The goal of the system is that the raw data can be available shortly after being collected. An acceptable delay is around a couple of minutes.

2. Feeding raw input data in batches:
   In addition to streaming data in real time, the system should be able to handle feeding data in a batch mode. A batch file can contain thousands of tags and their values in a time range. All features of these data samples are the same in feeding data in real time.

3. Aggregating raw input data into averaged data:
   Collected data can be stored up to ten years, and loaded for visualization on the user interface. It is such a massive amount of data that it cannot be processed within a few seconds. As a result, average of data points with different time levels such as ten seconds, one minute, ten minutes and one hour will be used instead. To support this feature, the system has to automatically pre-calculate averages of the data for a fast visualization for the user. However, aggregation is not required in real-time as recent aggregated data could be calculated from the raw data.

4. Dashboards:
   Dashboards of the system should visualize the current state of monitored measurements. The number of measurements required to display simultaneously in a dashboard should be several hundreds. Generally, the time interval for updating new values for each of the measurements may be from 10 seconds to 1 minute. In some specific cases, update rate can be slower, depending on the time interval of the process. However, these time intervals are not longer than eight hours typically.

5. Displaying raw data in a trend:
   In a user interface, application can visualize the values of a measurement as a trend based on the measurement being observed and range of time the user provides. The number of data points to display may be large, and reach up to 1 million data points. Therefore, the user interface should be able to show the trend on a screen which has the number of pixels less than the number of data

points. In addition, to guarantee a smooth and fast user experience, the query speed should be 1 million data points per second.

6. Displaying averaged data in a trend:
   This use case is similar to the previous one. Only the data for visualization is different. While the previous use case visualizes raw data, aggregated data (averaged) with different time levels such as ten second, one minute or ten minutes will be displayed.

7. Calculating results from a single column of data:
   The data of a measurement will be selected for selected time range. This data becomes the input of calculations such as descriptive statistics (average, variance and median), autocorrelation, histogram, spectrum or formulas provided by the user. The results should be available within one second for amount of one million data points or longer in case of expensive calculation. Also, the data the user selected can be optionally displayed.

8. Calculating the results between pairs of columns of data:
   In this use case, the data from a pair of measurements are chosen for a selected time range by the user. The amount of data can be large, up to one million data points. Then, the data is used for calculation like a cross correlation or any formula provided by users. Like the previous use case, the latency of displaying results should be less than one second for one million data points or longer with more expensive calculations. Optionally, the trend of data can be indicated in the user interface.

9. Search a single column of raw data for a time range based on the value of data in a piecewise constant measurement:
   The system provides users with a feature of finding all time ranges when a piecewise constant has a particular value. The requirement of executed time is within few seconds.

10. Finding measurements belonging to a category (grade, lab, dead):
    The system can classify measurements into different categories. All criteria for categorizing measurements are considered in a particular period and based on raw values. There are various types of categories, such as a group of measurements having no variation, containing discrete points, having no values in the required time ranges or fulfilling some criteria by the user. The results should be available within few seconds.

11. Finding control loops that fail validation:
    A control loop is a set of measurements and it works together in a process. There are different algorithms to validate whether a control loop works correctly or not. The system should provide the users with a list of control loops that fail invalidation process during a given time range defined by a user. The results should be available within few seconds.

12. Finding measurements with high correlation with the current measurement among all the measurements in the system:
    Initially, a user chooses a target measurement. In the next steps, the system has found all the measurements which have high correlations with the selected measurements in a given period. Notice that the period could be restricted by several configurations such as days, months and work shifts. The scope of the search could be a set of set of measurements user has defined or all the measurements currently in the system. With the target of one million channels, it will be a challenge to calculate high correlations among a full set of channels.

13. Finding measurements with similar spectral peaks as the current measurement among all the measurements in the system:
    Similar to the previous use case, the system can be able to provide a user with a list of measurements having similar spectral peaks to the chosen one, instead of high correlation during a period. The periods are pre-configured as days, months and work shifts. The data set for searching can be pre-defined by users or it can be the whole collection of measurements. The results should be available within a few seconds.

1.2.2 Overall analyses

Based on these requirements and use cases above, a solution using Big Data tools and frameworks in the Hadoop ecosystem is considered.

1.2.3 Thesis Structure

The rest of the thesis is organized as follows: Section 2 describes the general picture of Big Data and frameworks as well as several specific candidates for the prototype of the system. In the next section, the proposed architecture of the system is presented. Section 4 shows the performance of the system through several experiments. Section 5 briefly discusses results of experiments and evaluates whether the performance of the proposed system meets previous demands or not. Section 6 discusses the future work and conclusions.

2 Background

2.1 Overview of Big Data

These days, data is generated from almost all human daily activities such as uploading videos to YouTube, sending messages to friends as well as buying flight tickets, and the amount of data becomes increasingly larger. According to a report [10], the amount of data will double every year from 2012 to 2020. There are three main features of Big Data, including Volume, Velocity, and Variety (3V) [19].

- Volume: The amount of data is produced from daily people's activities and devices.
- Velocity: In some fields such as stock markets or sensors in cars, the speed of data generation is high, and it is required to make quick decisions based on the data. Therefore, processing this fast data in real time or in near-real time is focused as well.
- Variety: Data coming from a variety of sources such as signals from satellites, data from IoT devices connected to the Internet and data posts on social networks and in different formats including structured data (traditional SQL database), semi-structure data (XML or JSON) and unstructured (videos or audio).

The features of Big Data in the Process Industries
In the process industries, data collected from sensors in process lines or mills have specific features, such as

- Data type is time series type, meaning that all of the data collected include time values when they were generated.
- Majority of log data is structural. Also, there are other unstructured log types such as photographs taken in the process. However, in this master thesis, unstructured data is not considered.

2.1.1 Frameworks and Infrastructure

Storing and mining the massive amounts of data can generate useful information for science as well as for business. Therefore, recently popular tools, frameworks and infrastructures for storing and analyzing big data is described as follows:

Scalable Datastore (NoSQL Databases)
The strong growth in the volume of data also has resulted in the development of new kinds of datastores that are different from SQL databases. The meaning of NoSQL Databases is "Not Only SQL". NoSQL Databases provide new storage and access mechanisms, rather than relational tables in traditional SQL databases. Those features enable a NoSQL system to scale out horizontally up to thousands of nodes. There are various types of NoSQL datastores, enumerated as below:

- Wide Columnar Store: In the column family, the rows of a table are sparse, and the number of columns is not defined beforehand. It means that the number of cells in different rows can be different [25].
  HBase - A distributed and scalable database, inspired by Google BigTable [5], is built on top of Hadoop Distributed File System (HDFS). It provides random read and write operations to access data [26].
  Cassandra is a distributed database system in a massively large scale without a single point of failure point[16]. Cassandra uses some concepts and ideas from Google's BigTable [5] and Amazon's Dynamo [8].

- Key-Value Store: A Key-Value store works as a simple mechanism of a map/ dictionary model: retrieving or updating a value by the key. As recent key-value stores focus on high scalability rather than on consistency, they do not support analytic features such as joins and aggregates [25].
  Redis - an in-memory database with the key-value data model. It supports to store not only binaries but also lists and sets directly [25].

- Document Store: In a document database, data is organized as a collection of semi-structured document such as XML or JSON [3].
  MongoDB is a schema-free document database, storing documents in the BSON format which is similar to JSON [3].

- Graph Database: In a graph database, data and data model are stored and presented by a graph of nodes, edges and properties [1].
  Neo4j is a high-performance graph databases written in Java, and it uses Cypher, similar to SQL to query data [14].

Data Processing Frameworks
There are several distributed data processing frameworks for analyzing a massive amount of data to generate useful information, such as frameworks including Apache Hadoop, Spark and Flink.

- Hadoop MapReduce is an open source data processing framework, applying MapReduce paradigm, inspired from Google MapReduce [7].

- Apache Spark is a general data processing platform, a successor to MapReduce. Spark includes several built-in modules for different demands such as SparkSQL for exploring structured or semi-structured datasets, Spark Streaming for processing streaming data, MLlib and GraphX for sophisticated analytics in machine learning and graph processing respectively [31].

- Apache Flink is a streaming data processing engine, providing calculation on the flow of data [4]. While streaming data flow in Spark is the micro batch of data, streaming data flow in Flink is real-time data.

Big data SQL Framework
Some big data solutions contain frameworks for exploring structured data through a user-friendly SQL interface.

- Hive is a solution for manipulating data through SQL interface for Hadoop ecosystem. It was equipped with a query language similar to SQL to query and analyze data[27].

- Impala is an MPP SQL engine for the Hadoop ecosystem [15]. Impala provides high performance on datasets fitting in the main memory of the servers.

Cloud Services
Storing and processing a massive amount of data needs not only powerful frameworks or algorithms but also infrastructure such as the cluster of computers. Cloud computing, providing users the ability to scale their service with a lower price compared to deploying to their own cluster, is a solution for Big Data applications. There are several public clouds, such as Amazon's AWS and Google's Google Cloud.

In next sections, several potential frameworks and engines are discussed for evaluating in this thesis. They have a large number of users and considerable momentum behind them.

2.2 Big Data Processing Engine - Spark

Apache Spark is a next general distributed data processing platform, succeeding MapReduce. Spark executes computing in parallel on a cluster of commodity hardware, but still, ensures auto-scheduling, fault tolerant and load balancing [31]. Spark is developed in Scala, and exposes a functional programming interface.
In the first experiments, Spark outperformed MapReduce on a machine learning workload with 10x faster speed [31] and it sorted 100TB of data in 23 minutes with 206 EC2 i2.8xlarge machines in 2014[2]. Spark can achieve a better performance than MapReduce because Spark stores data in memory while MapReduce stores data on hard drives. More details will be described in next sections.

Spark Architecture
Spark architecture consists of a master node (driver) and several worker nodes (executors), illustrated in Figure 2. In each Spark application, there is a driver program (Spark Context Object) playing the role of a master node. Also, when running on a cluster, a Spark driver has to connect to the Cluster Manager, such as Yarn or Mesos to acquire resources for its applications. After being allocated several executors, the master sends application code (Java executable code or Python file) to the executors. Finally, the driver program continues to send tasks to the executors and waits for results from them.
Currently, Spark can run with three types of cluster managers:

- Standalone mode: A simple cluster manager shipped with Spark to build a cluster easily.

- Yarn: The resource manager in second generation Hadoop clusters [28].

[2]http://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html

– Mesos: The general cluster manager can run on various clusters of frameworks such as Spark and Hadoop[13].

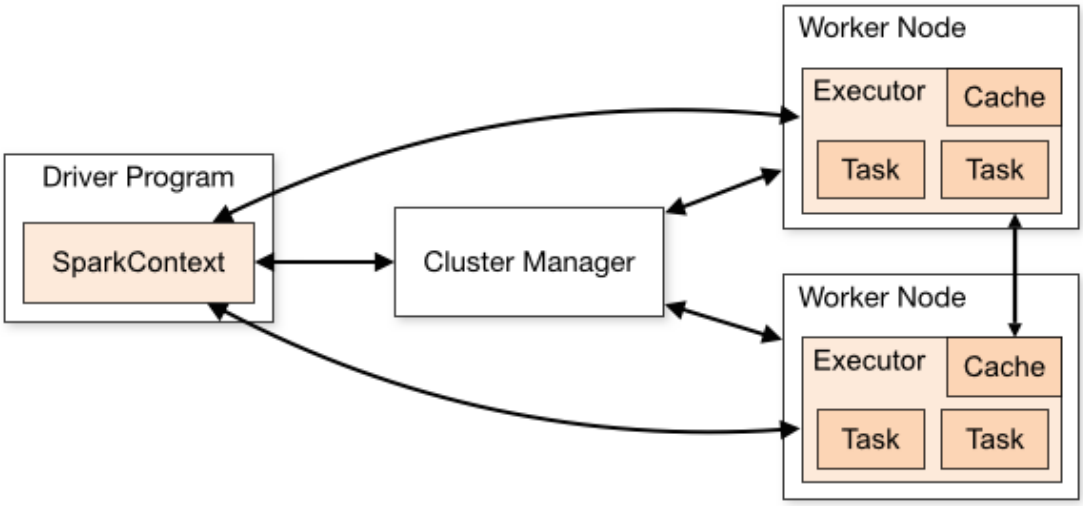

Figure 2: The architecture of Spark[3]

RDD, DataFrame and Dataset
Currently, Spark contains three different abstracts of datasets, including:

– Resilient Distributed Datasets (RDD) is the abstract of immutable datasets, distributed among nodes in a cluster. RDD provides interface API to developers to handle data in parallel [30]. Dataset of RDD is unstructured.

– DataFrame is an immutable distributed collection of data, and it has structural organizational analogy to a table in a relational database. DataFrame exposes SQL interface as well as a specific API to manipulate the dataset. More details about DataFrame will be presented in next sections.

– Dataset is a new abstract layer for immutable distributed collection of data in Spark 2.0. Also, from Spark 2.0, DataFrame became a part of dataset: a collection of objects Dataset[Row] with Row is untyped JVM object. Dataset has another strongly-typed JVM objects toward the classes defined in Java or Scala.

Benefits of the Dataset API: There are several advantages when using DataFrame and Dataset API, including:

– Runtime type-safety
In SQL queries, if there are some errors in queries, such as syntax errors, these errors will only be detected at runtime. With DataFrame and Dataset, the

[3]http://spark.apache.org/docs/latest/cluster-overview.html

compiler can catch syntax errors at compile time. With analysis errors such as a non-existent column, DataFrame cannot catch them in compile time, but Dataset can due to its JVM typed objects.

– Performance and Optimization
Due to the fact that DataFrame and Dataset API are built with Catalyst, the optimizer of the Spark SQL engine, programs written using these APIs can be optimized with it and executed with better performance than those written for the RDD API.

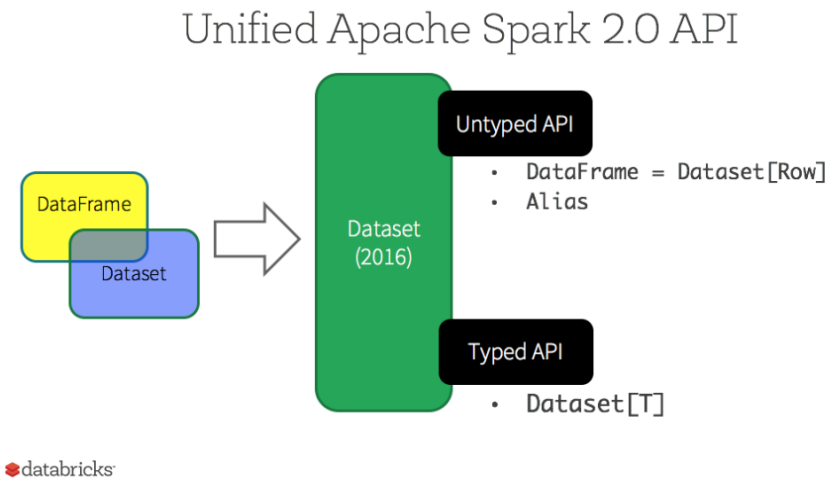

Figure 3: Dataset in Spark 2.0[4]

Transformations and Actions
In Spark, there are two kinds of operations on a dataset. Those are transformations and actions. A transformation is not executed at the time it is being called, while an action and previous transformations are carried out at the time the action is being called. This means that all transformations are only processed whenever an action operation is being called instead of being executed individually. Spark has a logical plan of all transformations and an action operated on a dataset. It then transforms that to a physical plan, run and return the final results. For instance, several transformations often used are 'select', 'groupBy', 'sum' and 'filter', and some often used actions are 'show', 'count', 'collect', and 'save'.

Spark's Ecosystem
In addition to the general processing features, Spark has included built-in modules used for different purposes such as SQL querying, data streaming, machine learning and graph processing. They are SparkSQL, Streaming, MLlib and GraphX.

[4]https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Spark SQL, one of the major modules, is used for processing structured data. This module was developed to adapt the requirements from customers such as data scientists and business intelligence users who prefer using SQL queries to explore data. The central component of SparkSQL is the DataFrame, a programming abstraction of structured datasets. With a DataFrame, users can use relational queries or a procedural API to achieve results. Moreover, Spark SQL is not only used for exploring data by queries but it also has a (combined) integrated machine learning module to do more advanced analyses.

Spark Streaming: Recently, Applications are required to adapt to the changes in the surrounding environment in real time. For example, a monitoring system has to raise an alarm whenever there is something strange happening. The batch processing approach is not suitable in this case. This requires an application to process data when it has just arrived. Therefore, Spark Streaming was developed with a feature to process micro batch data (nearly realtime) but still inherit advanced features from Spark.

MLlib: Machine learning has become increasingly important in exploring data to gain valuable insights. MLlib was built on top of Spark. It can easily leverage the computing power from clusters to execute complex algorithms with higher accuracy and speed.

GraphX is a built-in engine of Spark. It transforms, executes and explores graph data structures.

Project Tungsten
Project Tungsten was first introduced by DataBricks in 2014, and it was initially integrated to Spark 1.4. Last year, the second generation of Tungsten was shipped with Spark 2.0. Project Tungsten aims at efficient usage of memory and CPU in Spark applications. A recent research [23] pointed out that CPU and memory are the bottlenecks for Spark applications, rather than disk IO or network bandwidth. When network communication bandwidth reaches 10 Gbps, a high-speed SSD was used instead of HDD, and optimization of Spark on skipping unneeded data to minimize the effect of IO on transferring data, the performance of a task was only limited by CPU and memory efficiency. To tackle those bottlenecks, Databricks, the company behind Spark, introduced three optimizations in Tungsten, including Memory Management and Binary Processing, Cache-aware computing and Code Generation[5].

Memory Management and Binary Processing
The JVM performs all tasks on Spark, so these processes used JVM objects and relied on JVM Garbage Collector for managing memory. However, both JVM's objects and Garbage Collector have drawbacks on performance themselves. For example, a string with four characters needs 4 bytes to store in UTF8 encoding. However,

[5]https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

a JVM string having the same length will take 8 bytes for storing in UTF16, 12 bytes for additional header and 8 bytes for a hash code [6]. Moreover, JVM garbage collection (GC) is not optimized for managing memory for Spark applications. In JVM, there are two types of objects: short term and long term. Short term objects are allocated and deallocated in high frequency, and long term ones are maintained for a long time. GC relies on the estimation of the life cycle of the short-term objects to manage memory effectively. In several cases, this estimation may fail when an estimated short-term object is turning out to be a long term. Also, a Spark application may know the life cycle of their objects more clearly than JVM GC through a plan of computation and jobs and tasks in the plan. Therefore, Spark came up with a new solution of a C-style memory manager based on 'sun.misc.Unsafe' from JVM. The new memory manager controls directly binary data rather than Java objects. It means that the approach adds less overhead to objects and is invisible to GC.

Cached-aware Computing

As mentioned above, Spark overcomes Map Reduce by using memory based solution for analyzing data rather than using hard drives. However, Databricks [7] found that a large amount of processing time is to wait for the data fetched from RAM. To limit the drawback of main memory access, CPU caches, providing a higher accessing speed than RAM, are considered. Cache-aware computation focuses on leveraging the CPU caches L1, L2 and L3 for improving Spark's application performance. It contains cache-friendly algorithms and data-structures, supporting Spark applications to reduce execution time for waiting fetched data and spending time for performing useful tasks. As a Spark job is a set of operations such as sort, join and aggregation, so the improvement in these operations will promote the general performance of the Spark application. For example, sorting a collection of string records is a traditional way to

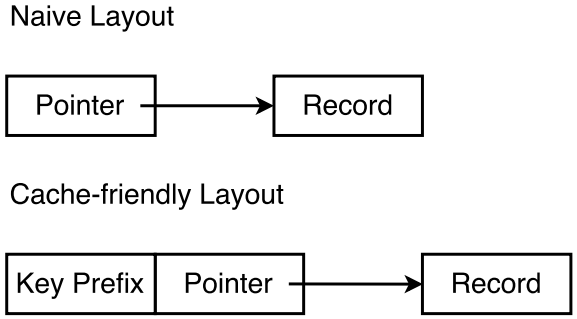

Figure 4: AlphaSort-style prefix sort[8]

swap the positions of two pointers by accessing memory and querying and comparing

[6]https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[7]https://databricks.com/

[8]https://spark-summit.org/2015/events/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal/

their values. Memory accessing consumes a significant fraction of the amount of time. So, to optimize the sort algorithm, a key prefix of values (the two first characters of a string, for instance) will be stored together with the pointer in the cache memory, illustrated in Figure 4. Storing key prefix with pointers will help comparisons between pointers executed in cache memory, instead of looking data in the main memory.

Code generation
In addition to code generation for expressions in SQL queries and DataFrames, which will be covered in next session, project Tungsten aims at using code generation for efficient usage of CPU. For example, with code generation, data serialization throughput is two times faster than with the Kryo serializer [9].

2.2.1 SparkSQL

SparkSQL is the most actively developed library of Spark currently [2]. SparkSQL is developed to adapt users' demands in using Spark for exploring and analyzing data through SQL interface.

The need of SparkSQL
Firstly, data sources vary from structured, semi-structured to unstructured ones. However, previous relational processing interface only supported the first type, the two former ones required custom code to handle. Secondly, in addition to common queries, sophisticated analytics such as machine learning also interest users currently. However, these tasks are impossible to be executed on relational process interfaces. In addition, users have to choose one of the two paradigms, relational or procedural, and then to integrate the remaining one manually. SparkSQL was developed to integrate two paradigms allowing users to leverage both benefits from the two paradigms without spending time for combining and tuning process.

To achieve these goals, firstly, SparkSQL provides DataFrame API that can explore data by relational process on both internal collections and external data sources like text and Parquet files on HDFS. DataFrame is an abstract of structure dataset which is manipulated by the Spark procedural API or the new relational API which has allowed to achieve better optimization. Secondly, in order to support various data sources and complex machine learning algorithms, Spark SQL also uses an optimizer named Catalyst. By using this Optimizer, users may gain various features such as inferring the schema from JSON and retrieving data from external databases. SparkSQL was built as a library that works on Spark and provides an interface working with JDBC/ODBC, console and DataFrame API, in Figure 5.

In the next sections, we will go on to find out more details of two main components of Spark SQL, including DataFrame API as well as the Optimizer Catalyst.

DataFrame API

[9] https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

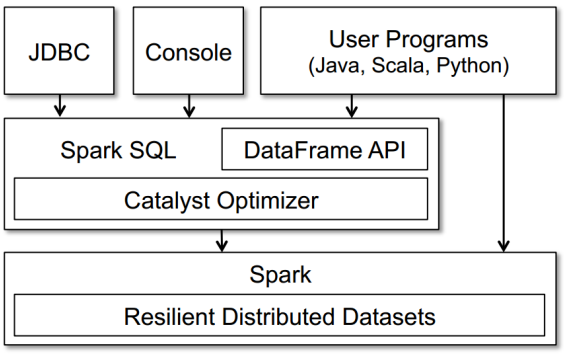

Figure 5: The Interface of Spark SQL[2]

As mentioned above, DataFrame is the immutable collection of data, like RDD, and it can be equivalent to an RDD of Rows. However, unlike RDD, each DataFrame contains a schema about its data structure and can be considered as a table in relational databases. A DataFrame is constructed from externally structural dataset such as Hive Table, CSV and JSON files, or from existing RDD. Also, DataFrame supports various standard SQL operations such as select, where and join and other expressions from different languages such as Python and R. These operations are considered as an abstract syntax tree for optimization by Catalyst later on [2].

Similarly to RDD, all transformation operations of DataFrames are "lazy", as they are postponed until a next action is performed. Each DataFrame contains a logical plan how to create and transform to current Dataframe. However, the plan is never applied until results of computations on the DataFrames are demanded. At that point, based on its logical plan, DataFrame will build a physical plan to calculate the outcome. Also, these relational operations and expressions of computation would be passed to Catalyst for optimization.

To support users familiar with the relational database, DataFrames provide users with the ability to achieve results using SQL queries by registering a DataFrame as a temporary table. However, DataFrame is more convenient than SQL queries due to the integration of DataFrame and programming languages, including Java, Spark, Python and R. For instance, immediate results of queries can be achieved and debugged.

Moreover, Spark DataFrame also has several features such as Querying native Datasets and In Memory Caching. One of the most useful features SparkSQL provides to developers is to query native datasets. In practice, there are heterogeneous data sources operated by different libraries. To work smoothly with SparkSQL functions, Spark enables developers to construct a DataFrame directly from RDD of native datasets, and automatically infer the schema of the dataset from the RRD. With

statically typed languages such as Java and Scala, SparkSQL obtains the type of attribute from language system's type, while with dynamic languages such as Python, SparkSQL bases on random values of the dataset. When SparkSQL executes a query, it makes a logical data scan operations on an RDD, transforms them to physical operations, then accesses the native dataset in places, and only retrieve data required for the query. Also, under the format of the temporary table, RDD of a native dataset can be combined with external structure data sources, such as Hive tables. Regarding In Memory Caching, Spark SQL caches data in columnar format, which reduces the footprint of memory by order of magnitude due to compression schemes such as run-length encoding and dictionary encoding.

Catalyst Optimizer

In SparkSQL 2.0, the most important component is Catalyst, which does optimization on all of the queries before executing. In the past optimizer frameworks such as Cascades or EXODUS [6, 12], optimization was built in a complex domain-specific language, then a particular compiler compiled the optimized source code to executable code. In Catalyst, Scala is chosen as the primary language to build specific rules of optimizers, lead to convenience in maintenance and debugging. Building a new optimizer, Catalyst aims at two goals. Firstly, Catalyst provides an interface which is convenient for adding new optimizations for handling different kinds of semi-structured data sources or tuning algorithms for advanced analytics. Secondly, Catalyst enables external developers to build or add more features to optimizer libraries for new data types or rules applied to external sources, for instance. Therefore, Catalyst is written in Scala, allowing developers to extend optimizers easily [2].

The catalyst consists several libraries for presenting objects on source code as a tree and collections of rules that applied to the tree for tuning. The tree presented to operators and expressions in query processing, while the set of rules are implemented in various stages of query execution.

Tree
A tree in Catalyst is composed of nodes, having types defined as subclasses of TreeNode class in Scala [2]. For example, the expression x + (1+2) can be presented as a tree in Figure 6. The nodes of the tree are defined as these classes below [2]:

- Literal - presents a constant value.
- Attribute - presents an input variable.
- Add - the sum of values in two branches of the node.

Rules
In Catalyst, functions applied to trees and transforming those trees to the other trees [2]. Typically, a rule is a pattern matching function, which is applied to all nodes of tree recursively, and it transforms the type of those nodes. In reality, rules

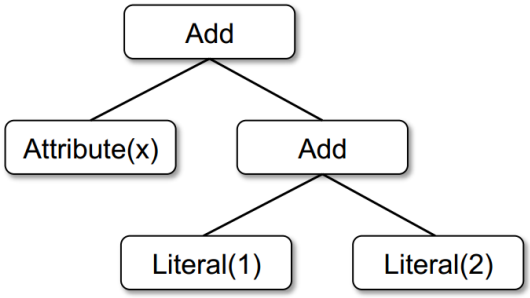

Figure 6: The illustration of tree of the expression x+(1+2) [2]

are performed on a tree many times, until the fixed point at which the tree does not change anymore by the rules. For instance, in the example above, the expression $x + (1 + 2)$ will be applied to the rule of constant folding [21] and transformed to $x + 3$.

Phases of Catalyst optimizing queries in Spark SQL
In practice, the Catalyst optimizer is illustrated in Figure 7, composing four phases, including analyzing an unresolved logical plan, optimizing the logical plan and building physical plans and generating Java executable code. Three first phases are based on rules, while the last one, to select a plan and execute bytecode, based on cost. The details of phases will be presented in next sections.

– **Analysis**

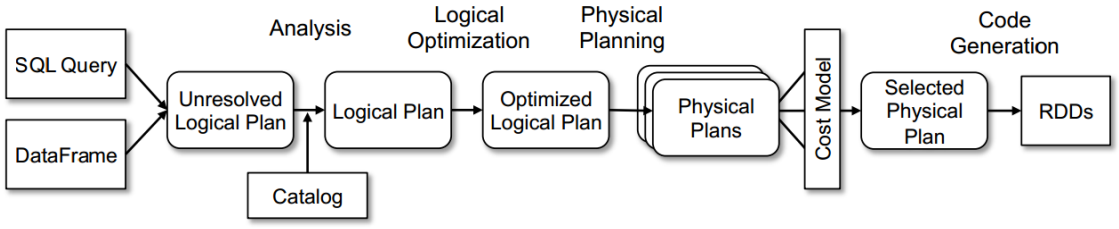

Figure 7: Phases of Catalyst optimizing queries in Spark SQL [2]

The input of Analysis phase can be the Abstract Syntax Tree (AST) resulted from an SQL parser or DataFrame, which is created by API [2]. Both of them may contain unsolved attributed references, having unknown types or not matching with any field in the input table. For instance, in a SQL query, the user may look for a column which does not belong to a table or the type of column is unknown. Therefore, to resolve logical plan, optimizer has to look up the names of the attributes in the catalog which store the metadata of the table, then map table's fields to input's attributes.

– **Logical Optimization**
The logical plan is applied for rule-based optimizations. Catalyst uses several optimizations logic like compilers. For example, constant folding is calculate constant expressions and predicate pushdown is to delivery conditions and execute them to dataset instead of load the whole dataset then filter. In Spark 2.0, Catalyst enables external developers to add their optimizers.

– **Physical Planning**
In this phase, Catalyst generates several physical plans from the logical plan in the previous steps. That physical plan contains operators that can be executed by Spark engine. The selection of the physical plan will be based on the cost model. In addition, the physical plan also implements physical optimizations based on rules such as pipelining filters in a map operation [2].

– **Code Generation**
After selecting a physical plan, Catalyst continues generating Java executed code from the plan. To make the compilation of code faster and simpler, Catalyst uses Scala compiler with a particular feature being quasiquotes. Quasiquotes are Scala snippets using string interpolation syntax, supporting manual Abstract Syntax Tree (AST) construction and deconstruction which are error-prone [24]. Therefore, Catalyst will convert a tree of SQL expressions or operation to AST with quasiquotes; then a Scala compiler to produce Java bytecode from the AST [2]. Also, in the phase, Scala compiler will continue checking and executing optimizations maybe missed in previous rule-based phase.

Performance after booting by Whole-Stage Code Generation (Project Tungsten)
In Spark 2.0, "whole-stage code generation", based on ideas borrowing from several modern compilers and massively parallel processing (MPP) databases [22], is a part of second generation Tungsten engine. Whole-stage code generation aims to improve Spark application's performance by combining the entire query to a function for optimizations such as eliminating virtual function calls, saving intermediate data in CPU register rather than main memory. Volcano [11] model is used in the previous version of Spark as well as many other database systems. In Volcano model, a query contains a set of operators which are performed consequently. By dividing a query to a set of composable elements, all queries can be presented by combining different operators without a fear of type mismatch between these operators. However, this model is not optimized. Whenever data is passed between operations, dynamic dispatch process is called. It takes several CPU instructions. Also, immediate data passing between operators are forced to write to main memory. The overall process slows down by fetching data from the main memory. In contrast with the Volcano model, particular few lines of code in Java and Scala for specific tasks is not certainly composable but is optimized for these tasks and achieve better performance. For example, it does not dispatch virtual functions, and intermediate data is placed directly into a CPU register by the compiler. Whole stage code generation aims at

achieving the performance of the direct code. From the results of benchmarking of Spark 1.6 and 2.0 in Figure 8, using whole stage code generation gains a better performance in Spark 2.0 compared to Spark 1.6.

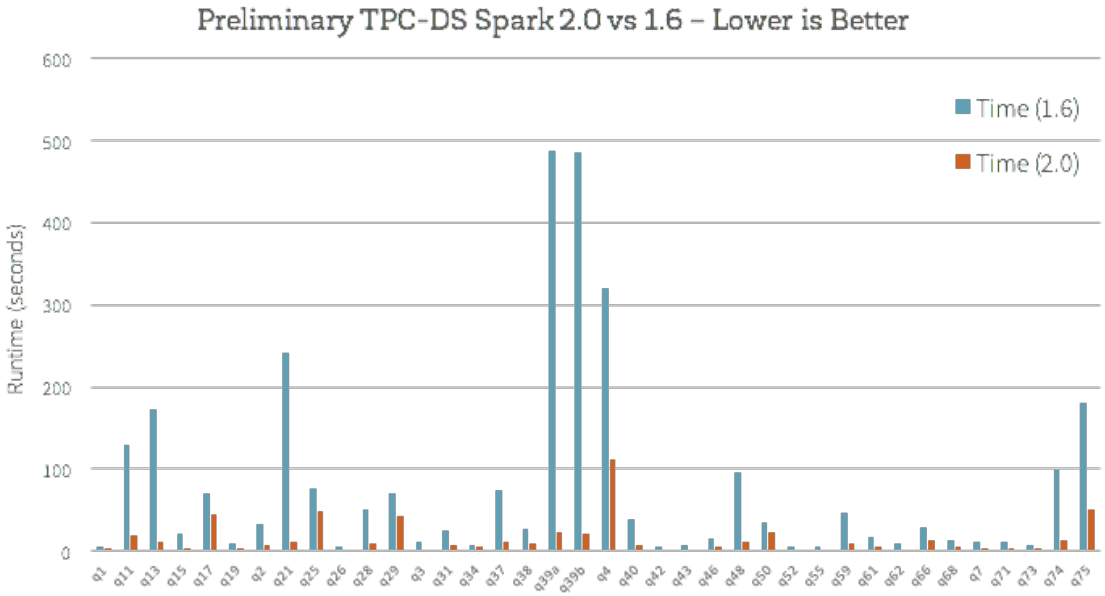

Figure 8: TPC-DS queries to compare Spark SQL 1.6 and 2.0[10]

2.3 Data File Format - Parquet

Parquet is a columnar file format in the storage layer of Apache Hadoop. Parquet is also compatible with most data processing frameworks in Hadoop ecosystems such as Spark, Impala, Hive and Pig. It provides efficient data compression and encoded schema, achieving high performance in querying data. Parquet was inspired by Google's Dremel [20], and it uses Dremel's algorithms to represent nested schema to flat files.

Why does Parquet use Column format?
In relational database management system, data is usually represented in a table format. However, on disks, data is written sequentially, and the next row will be appended right after the current row (in the row layout). For example, in Figure 9, when data is queried, for instance, from all rows of column a, the read operation will first get data in position a1, then jump to a2 and so on. It seriously spends a lot of time in retrieving values of a column. Therefore, to reduce wasted time for seeking data, database management system should store values which belong to a same column in the same place. This format is called column layout. The difference between the row and column layouts is presented in Figure 9.

[10]https://databricks.com/blog/2016/07/26/introducing-apache-spark-2-0.html

Typically, the row layout is efficient to transaction queries, based on data in a row, and column layout is applied for analytics purposes, based on data in a column.

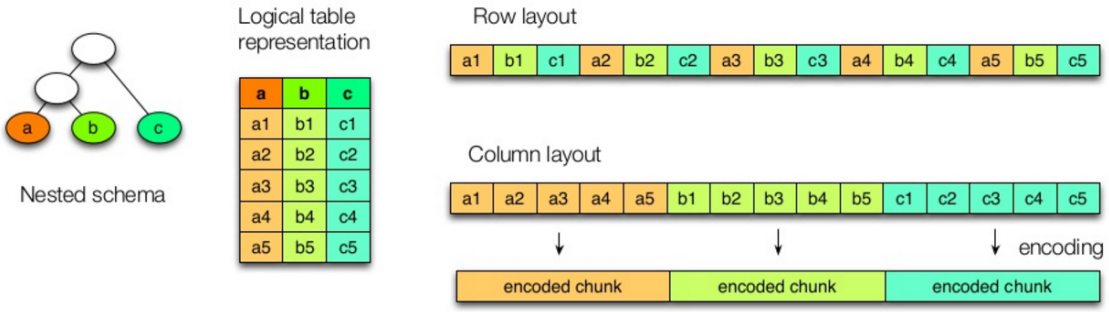

Figure 9: The representation of data in the row layout and column[11]

2.3.1 Parquet's Benefits

As a columnar file format, Parquet offers several advantages to analytic system, including efficiency in the storage space, performance of queries and support for multiple frameworks and tools.

Multiple framework's support
Parquet provides different adapters (drivers) working properly with various frameworks and tools in the Hadoop ecosystem, illustrated in Figure 10. Integration or migration from one application to another is easy and pleasant. There are several applications supported by Parquet, such as:

- Data processing frameworks: Spark and MapReduce.
- Query Engines: Hive, Spark SQL, Impala and Presto.
- Data Models: Avro.

Space Efficiency
When data is stored in a columnar layout, all data have the same type and will be placed in the same physical location. As a result, this can help a system to encode and compress data with a high ratio, providing less space to store than in a row-oriented layout. There are several encoding types which Parquet supports:

[11]https://www.slideshare.net/cloudera/hadoop-summit-36479635
[12]https://www.slideshare.net/cloudera/hadoop-summit-36479635

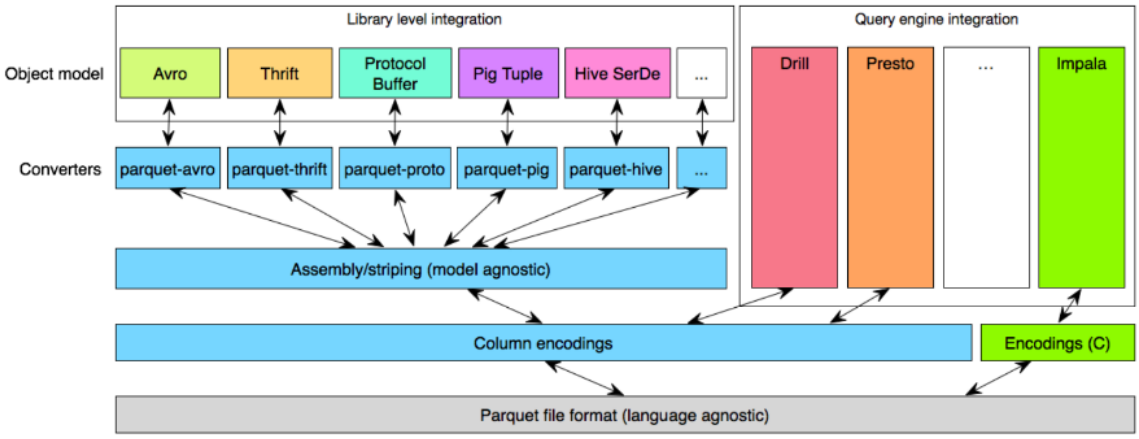

Figure 10: Parquet's internal operations and frameworks it supports[12]

Dictionary, Delta, Run Length and Plain encoding [13]. The plain encoding will be implemented when other efficient encoding methods cannot be applied. In addition to encoding data, Parquet also supports several compression types, such as gzip and snappy (default compression). In an experiment [14], 194 GB of data in CSV file format was compressed into 4.7 GB in the Parquet format. The compression ratio of Parquet thus reached 97.56%.

Query Efficiency
In particular, data in an entire column was usually selected for analyzing in OLAP applications. Instead of scanning all rows for retrieving data from several columns in the row layout, a reader only seeks in some particular segment of Parquet files. Also, Parquet reader supports various filters such as projection pushdown or predicate pushdown filters for eliminating data transferred between datastores and applications. Moreover, data is encoded and compressed in Parquet format and it also helps to reduce the amount of data exchanged. In an experiment [15], the performance of Impala, MPP SQL engine on Parquet file format was almost 4x or 5x higher than the same data in text format. Also, Parquet is self-describing, so its schema is preserved. It is convenient for loading data to frameworks such as in Spark, a Parquet file can be converted into a DataFrame or Dataset still maintaining its internal structure.

2.3.2 Parquet's structure

Although Parquet is a columnar file format, data in a Parquet file is split into row groups. A file can contain a single row group or multiple row groups, and the number of row groups per file can be modified. A row group includes each column chunk for a column of data in a table. Column trunks are organized in a columnar layout and

[13]https://github.com/Parquet/parquet-format/blob/master/Encodings.md - accessed on 5th June 2017

[14]https://blog.cloudera.com/blog/2016/04/benchmarking-apache-parquet-the-allstate-experience/ accessed on 5th June, 2017

they are contiguous to each other. In a column trunk, data is separated into pages, which are the smallest and undivided units, regarding encoding and compression. Also, a Parquet file contains a footer including metadata for row groups and column trunks. The structure of a Parquet file and its details is illustrated in Figure 11.

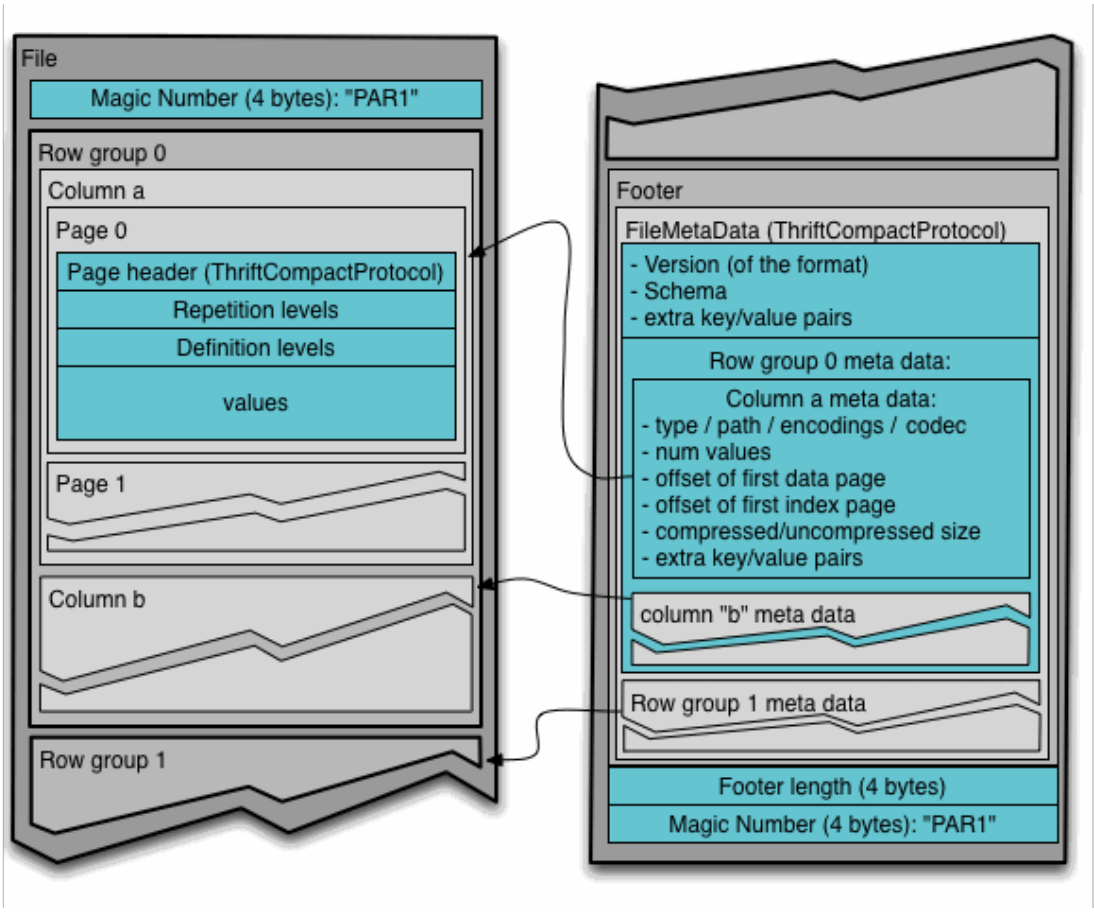

Figure 11: The structure of a Parquet file[15]

2.4 MPP SQL Engine - Impala

2.4.1 Impala and MPP SQL Engines

Impala is an MPP SQL engine in the Hadoop ecosystem [15]. Like MapReduce, in an MPP, data is distributed and processed across the cluster of machines. However, MPP database engines exploit the shared-nothing architecture, in which nodes have their memories and process quite independently, and they only work on the different partitions of the same dataset. These data processes are executed independently and in parallel, then the separated results from processors are collected and combined to the final result. As a result of implementing shared-nothing architecture, Impala, as

[15]https://parquet.apache.org/documentation/latest/

well as other MPPs in general, do not have bottlenecks on cooperation between nodes such as shuffling data in the MapReduce Paradigm. Therefore, Impala was developed to meet the demand of low latency queries for business intelligence (BI) or interactive analytics for the Hadoop environment, which batch frameworks such as Hive cannot provide. Through experiments in [9], Impala achieves a better performance than Hive, illustrated in Figure 12.

Impala, developed in Java (frontend) and C++ (backend), provides JDBC/ODBC interfaces for BI customers, as well as maintaining the flexibility of Hadoop ecosystem by supporting widely different file formats such as Avro, RCFile and Parquet [15]. Regarding the SQL support, Impala does not provide UPDATE and DELETE statements, because of the limitation in random write operators of HDFS as well as storage managers in the cloud environments such as S3. In next sessions, the architecture and internal operations of Impala will be explained in detail.

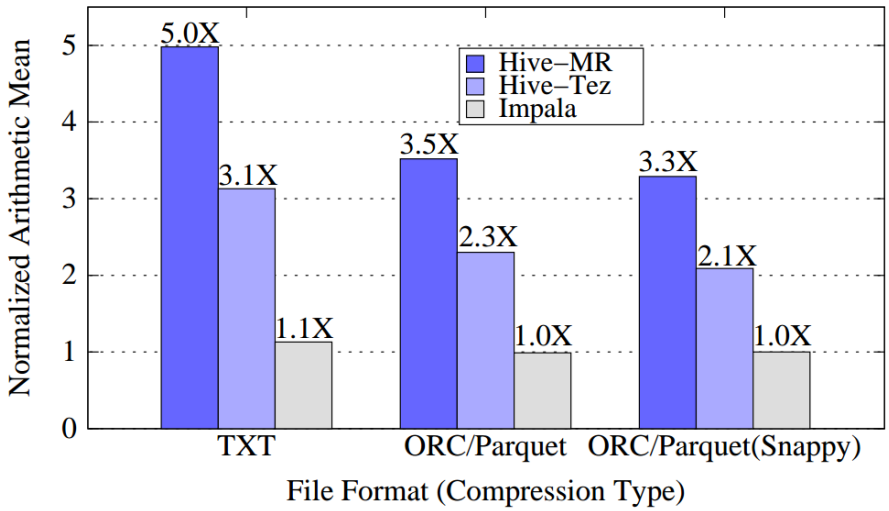

Figure 12: Comparison between Hive (MapReduce and Tez) with Impala in different format data[9].

2.4.2 Architecture

In overview, Impala consists of three various services, including Impala daemons (impalad), a state store daemon (statestored) and a catalog daemon (catalogd). Each of Impala daemons was deployed in each node in a Hadoop cluster, illustrated in Figure 13.

–Impalad

Every impalad is composed of a query planner, a query coordinator and a query executor like in Figure 13. The query planner and the query executor will be discussed in detail in the next session.

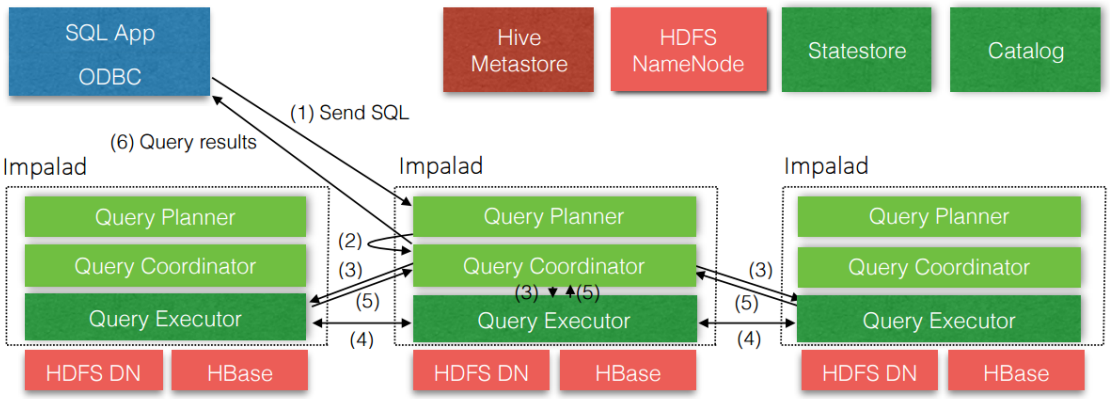

Figure 13: The architecture of Impala and flow of query processing[15].

The Flow of Query Process:
In a cluster, every impalad is equal to the others, and it means that any impalad can play the role of a coordinator. When a client needs to query Impala through JDBC or ODBC interfaces, a connection between SQL application and any impalad will be established. SQL queries will be sent to Query Planer, where it was parsed, analyzed and optimized. Then, the output plan is passed to the query coordinator of the impalad. The query coordinator distributes executable plan to the local query executor as well as query executors in other impalads with the goal of minimizing data movement. Query executors processed the chunk of a dataset in the same node, then returns the results to the query coordinator in the previous step. In the query coordinator, the outcome is combined and returned to the SQL application.

– **Statestored**
Statestored is a service for exchanging whole-cluster metadata with each Impala daemon. Statestored operates under the publish-subscribe mechanism, which pushes updates to specific subscribers. In the state store, there are a set of defined topics, containing entries - tuples of (key, value, version) [15]. These entries are changes of Impala metadata. Subscribers are processors want to receive updates from the state store service on some specific topics. Then statestored periodically send the updates of registered topics such as new, modified or removed data to subscribers. If a receiver wants to modify the list of items they want to follow, they will send back to statestored new lists after receiving an update from the state store service. Also, Statestored periodically sends keep-alive messages to subscribers. When a failed receiver is detected, Statestored will stop publishing data to the receiver. All time-out subscribers have to register the list of topic with Statestored from the beginning.

– **Catalogd**
Catalogd (Catalog service) works as a catalog repository for storing Impala's metadata. Whenever metadata in the catalog is modified, such as updating information on a new table, those updates are broadcasted to Impala daemon through state store

service. Also, the Catalog service also works as a metadata gateway for external data sources such as Hive Metastore and HDFS Namenode. For example, from the requirements of impalads, catalogd executes data definition language (DDL) operators to retrieve information from external data source's catalogs above. Then that information is combined with existing information in the Catalogd.

2.4.3 Internal operations

– **Frontend**
Impala frontend (Query Planner), written in Java, receives an SQL query from an application and builds an executable plan for the backend. Like traditional SQL planners, the query planning process consists of several steps, including query parsing, semantic analysis and cost-based query optimization. Also, its parser fully supports SQL operations such as select, join and group by. However, different to traditional SQL optimizers, the Impala query planner consists of two different plans, including the first plan for a node and the second one for multiple nodes.
In the first step, a parsed tree from SQL parser, combining some operators such as HDFS scan, hash join, sort and top-n will be translated into single-node plan tree. Based on information of dataset partitions, data offsets, Impala query planner will calculate predicates as well as perform cost-based optimization for minimizing the total evaluated cost.
In the second phase, the distributed and executable plan will be created from the previous single-node plan. The goal of this step is to minimize data movement as maximizing local data scanning by adding exchange and non-exchange nodes between planned nodes. Aggregate operations in this step are all local pre-aggregations. In the end, whenever results are collected then a final merge aggregation will be executed.

– **Backend**
Impala's backend (Query executor) is responsible for implementing distributed plan retrieved from its frontend. Impala's executor is written in C++ and apply runtime code generation for optimizing code execution and reducing memory overhead [15].
– Code Generation
As using LLVM [17] for code generation at runtime, Impala executor improved performance by 5x for typical workloads [29]. By performing just-in-time compilation (JIT), Impala code generator can deliver a specific machine code beneficial for performance by removing virtual function calls. Performance of Impala with the enable/disable code generation feature is illustrated in Figure 14.
— I/O Management
Initially, Impala supports storage in the same cluster such as HDFS or HBase. However, with emergence of cheaper and durable storage in cloud, Impala also supports exploring data in popular public cloud services such as S3 (AWS). Several recent versions of Impala also support reading/write to S3 directly without moving data to local storage [16].

[16]https://blog.cloudera.com/blog/2016/08/analytics-and-bi-on-amazon-s3-with-apache-impala-incubating/#primary

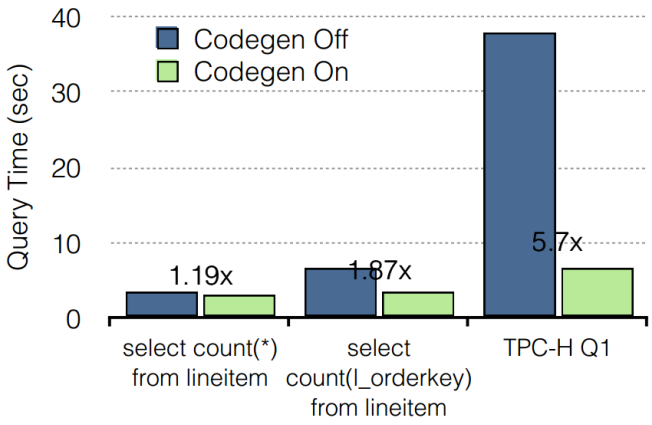

Figure 14: Impala's Performance when enable/ disable code generation feature[15].

– Storage Formats
To keep a flexibility in the Hadoop ecosystem, Impala supports several data formats in the storage layer such as Parquet, RCFile, Avro, Sequence and plain text, as well as compression techniques such as snappy and gzip. Parquet format is recommended with Impala for achieving better performance than other file formats [15].

3 System Design

This section presents the architecture of the prototype system using a platform and technologies mentioned above in detail, the design of the datastore as well as various tuning processes for those frameworks to enhance the performance of the system to meet the requirements from the use cases.

3.1 System Architecture

In addition to the main components mentioned in previous section, the prototype also contains other tools, including Sqoop, Livy and Bokeh. Parquet files are stored in S3, the Amazon Cloud Storage. The features and their functions are described in next sections. The whole system and the interactions between them are shown in Figure 15. The first group is responsible for collecting, storing and processing data (Sqoop, Spark, Livy, Impala, S3 - Parquet files), and the second group visualizes data and interacts with end users (currently web interface build on Bokeh library).

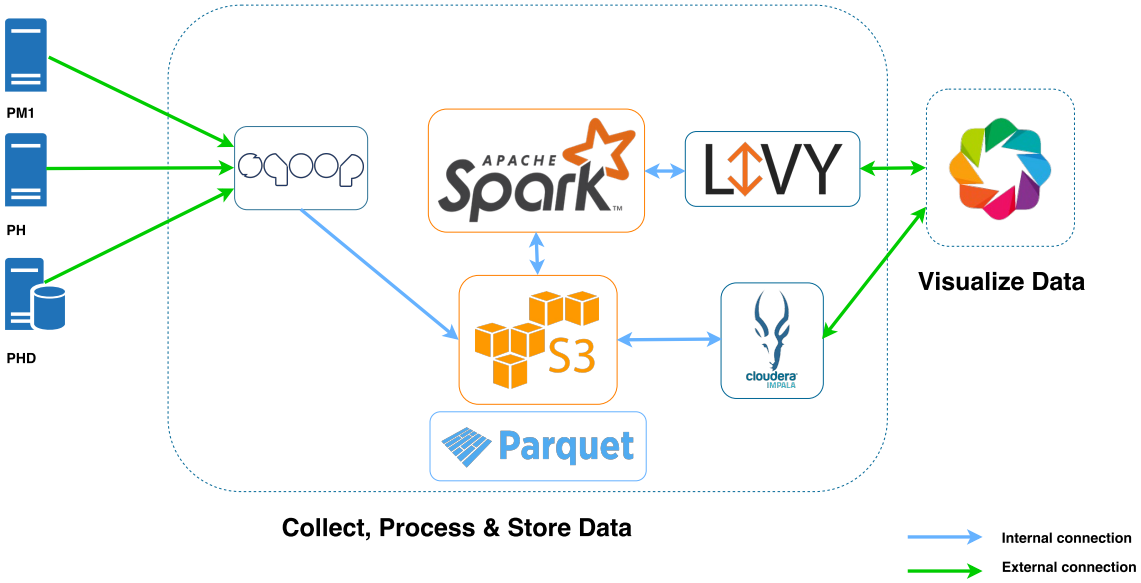

Figure 15: The architecture of system

– **Livy**
Livy is a service providing ability to interact with Spark through a RESTful interface. A user can submit a task to Spark through Livy in a batch mode or in an interactive mode. In batch mode, like spark-submit, the user has to upload a jar file to Spark, while in interactive mode, like spark-shell, the user can send snippets of Spark code[17]. Results can be returned in a synchronous or asynchronous mode. Also, Livy can manage several Spark Contexts at the same time, enabling the system to support multiple users with different Spark Contexts. However, Livy allows clients or jobs to

[17]http://livy.io/overview.html

share cached datasets with each other.

– **SQOOP**
At first, Sqoop [18] is an application that helps to exchange a massive amount of data among structured datastores like relational databases and unstructured datastores, such as HDFS. In the next generation of Sqoop (Sqoop2), the types of datastore are extended, including semi-structured databases, such as HBase or Cassandra with a relational database or HDFS. The internal process of Sqoop are based on MapReduce. For example, to import data from relational database to HDFS, Sqoop firstly collects the meta data of the relational database. From this kind of information, Sqoop splits data into chunks of data and uses Map function to import these chunks data to HDFS. Conversely, when Sqoop exports data from HDFS to a relational database, Sqoop gathers the metadata of the database, then applies Map tasks to pull the chunks of dataset in HDFS and imports them to relational database.

– **S3**
S3[19] (Simple Storage Services) is a fault-tolerant, distributed object storage which is designed for storing data for a long time with low cost. There is no directory in S3, all data is stored in different objects, named buckets. At the time this thesis was written, the maximum size of data to upload to S3 is 5 GB. In this architecture, S3 is responsible for providing data for analytics and storing raw data for a long time.

– **Bokeh**
Bokeh[20] is a interactive visualization library written in Python. In this project, bokeh is used to generate a graphics for a trend of data. Bokeh is chosen for building the prototype as Python has a powerful library - Panda, having structure like DataFrame in Spark. When data is de-serialized from a JSON object, it only requires some simple steps to convert data to DataFrame and to visualize on a web interface.

3.2 How does the system work?

Firstly, data from different sources, such as PM1, PH and PHD will be gathered and fetched by Sqoop several times a day, depending on the size of batch of collected data or requirements in using data for analysis. Data is converted to Parquet files by Sqoop, and instead of storing in HDFS, Sqoop will push these Parquet files to S3. When data was import to S3, Spark periodically checks to pre-aggregate new raw data, for example one second to aggregated data with different time levels, such as two minutes, ten minutes and one hour.

When a user wants to execute a calculation on data or see the trend of data, the user interacts with a web interface built on Bokeh library, then sends a request to Livy. In the next step, Livy checks whether any Spark Context in users' sessions is

[18] http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html
[19] https://aws.amazon.com/s3/
[20] http://bokeh.pydata.org/en/latest/

available or not. If there is not any available Spark Context, Livy creates a new one. In case a Spark Context is available, Livy will assign this task to the Context, Spark will execute the task and returns the result to the user through Livy. Note that, when a Spark Context is create, it will take some amount of time to start, having an impact on the user experience. Therefore, in case of selecting data for visualization (no complex calculations), Impala can be used instead of Spark.

3.3 Datastore Design

As being mentioned above, in order to increase performance of mining data later on, the collected data is stored as Parquet files. However, to achieve higher performance, data should be divided into different partitions. These help SQL engines to retrieve data faster as they can locate where data is and only read a certain amount of data instead of scanning the whole datastore. In the current system, users mostly look for data of several channels in a certain range of time. Therefore, the new datastore will be partitioned by different levels of time as well as groups of channels.

3.3.1 Partition by Time

Typically, time series data is formed in the format of 'YYYY-MM-dd HH:mm:ss'. However, with this format, data is difficult to partition into different levels of time. Therefore, time data is split into different fields, including Year, Month, Day and Time (in a day) in the new design of system. In addition to advantages of higher performance in querying, partitioning time data has other benefits:

- Remove expired data
  One of the requirements for designing a new storage schema is to be able to delete out-of-date data up to demand by year, month or even date. To fulfill this requirement, data should be divided into different options, including yearly, monthly and daily levels.

- Distribute data evenly
  By partitioning time, data is distributed evenly in chronological periods of time, for example, 2016 and 2017. If all yearly data is at the same place, the amount of data needed to scan for each query keeps on increasing and the performance of the application will be decreased.

3.3.2 Partition by Group of channels

Channels will be partitioned into groups, machines or processes which those channels belong to. This may help applications to increase performance when reading data of several channels relating to each other, for example, users are looking for several channels of a machine. Moreover, partitioning data by groups of channels can help the system:

- Avoid interruption when writing data
  As the memory for writing data is defined by the size of Row Group and the

number of opening files, the large number of files opened will crash the process by an exception 'OutOfMemory'. When being ingested into a datastore, data is also written according to a group sequentially. This helps the application to avoid opening a large number of parquet files to write at the same time[21].

The illustration of the datastore partition is shown in Figure 16. In the highest level, data is partitioned by year. In each year, data is continued to be split into different months. Then, data in each month is partitioned into different days. Finally, data is divided into various groups which its tags belong to. Partitioning helps to Parquet Reader to quickly find the location of data it is looking for. For example, if Parquet reader is looking for all values of Tag = 1, Group = 2 on 17 March 2017, it will find the location of Parquet file storing the data it needs. Also, instead of transferring a massive amount of unneeded data, the reader only pulls data in Group = 2, Day =17, Month =3, Year = 2017.

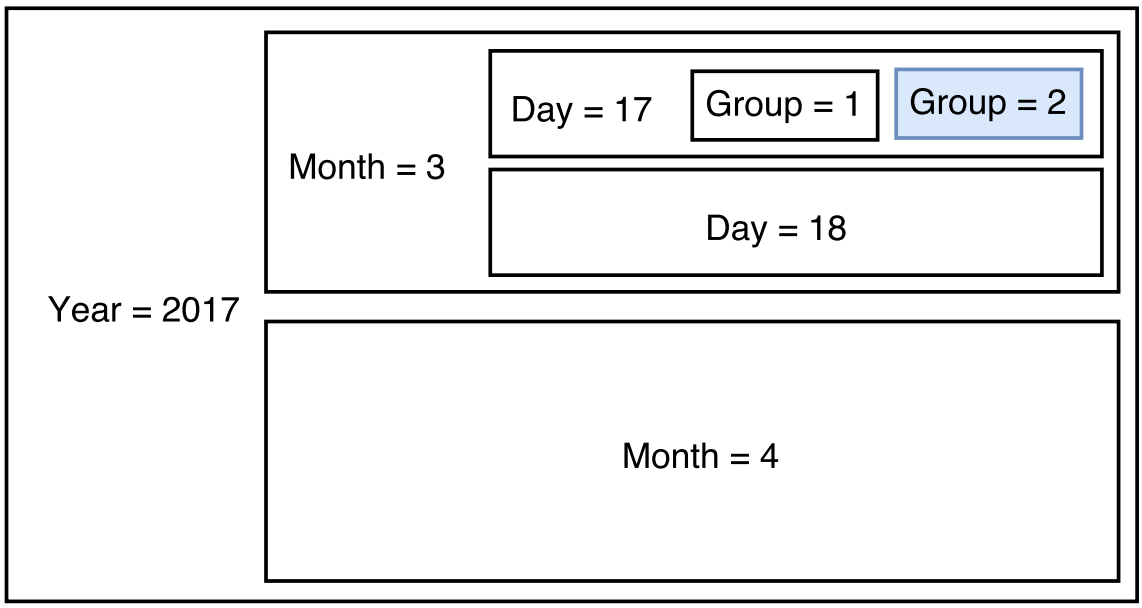

Figure 16: Illustration of the datastore partition

3.4 Tuning

3.4.1 Spark and Yarn

As mentioned before, when Spark is running in a cluster, there are three different deploy modes, including Standalone, Mesos and Yarn. In the experiments of the thesis, Yarn was used for managing the resources for a Spark cluster. To maximize the performance of Spark, we have to optimize the resources YARN allocates to Spark. Basically, from physical cores and RAM of a cluster, YARN combines these

[21] http://ingest.tips/2015/01/31/parquet-row-group-size/

resources to create many smaller containers. This can be archived through an option when starting a Spark application:

- executor-cores : the number of cores per executor.
- executor-memory: RAM of each executor.
- spark.executor.instances: The number of executors.

When Spark has multiple executors, the work is shared by different executors so that it can gain advantage. However, if each container only has one core, the performance of the system will be affected. As, with only one core, containers cannot leverage the benefits from multi-threading of JVM.

3.4.2 Impala

One of the methods to tune Impala's performance is to collect information about metadata of datastore, such as a distribution of data in a table, information's column. This can help Impala optimize queries when executing. This can be achieved by using a statement "COMPUTE STATS" with unpartitioned table and "COMPUTE INCREMENTAL STATS" with a partitioned table.

3.4.3 Parquet

There are several techniques to improve performance of writing or reading data from Parquet files, which can enhance the overall performance of the system.

- Increasing the size of a Parquet file or to reduce the number of small Parquet files:
  Firstly, data is collected and processed by Spark, then written to S3. By default, data is shuffled and processed by different spark's nodes, then it is divided to different partitions on S3. Then, in each new partition on S3, each of Spark workers write a tiny amount of data it has, leading to a large number of Parquet files for one partition. In total, there is a huge number of Parquet files on storage, with the size of each file quite small, for example, few KBs to few MBs. Due to effects of I/O and the network, reading thousands of small Parquet files will make system suffer and downgrade the performance. Therefore, before saving as parquet files on S3, partitions will be merged in Spark by 'coalesce' into smaller number of partitions with a condition that each output parquet file is large enough.
- Set Row Group size is the same as the size of the file (a Parquet file store a row group):
  As being mentioned above, A row Group is the largest section in a Parquet file, which groups data as rows. However, inside a row group, data is organized into several column chunks. Basically, data is retrieved by Row Group, then column chunks are selected on demand. If the size of the Row Group is small, the amount of data in column chunks is small, and it cannot leverage the Parquet's

column format. Conversely, if the size of a row group is larger than the size of a Parquet file, data in the group will be spread into two files. As a result, readers have to read both Parquet files for retrieving data of the row group instead of one files. Therefore, in order to gain a better performance when reading Parquets files, a Parquet file ideally stores only a row group.

– Turn on predicate pushdown filter:
To achieve higher performance on reading data from Parquet files, reducing the large amount of unnecessary data to scan and transfer is necessary, especially when data is stored in external storages such as S3, and transferred to application through network. Predicate pushdown filter may help to eliminate unnecessary data for queries. When a query contains WHERE clause, predicate pushdown filter is going to help the Parquet Reader to retrieve exactly the values that satisfy conditions in WHERE clause, instead of the all values in the columns selected. Figure 17 illustrated how Predicate Pushdown cooperate Projection Pushdown to retrieve exactly the required data. At first, the Parquet reader uses Projection Pushdown to choose only the columns selected in the query (such as column b in Figure 17). Then, Predicate Pushdown Filter will be used to select values which fulfill conditions of the query, for example values b2 and b3 in Figure 17).

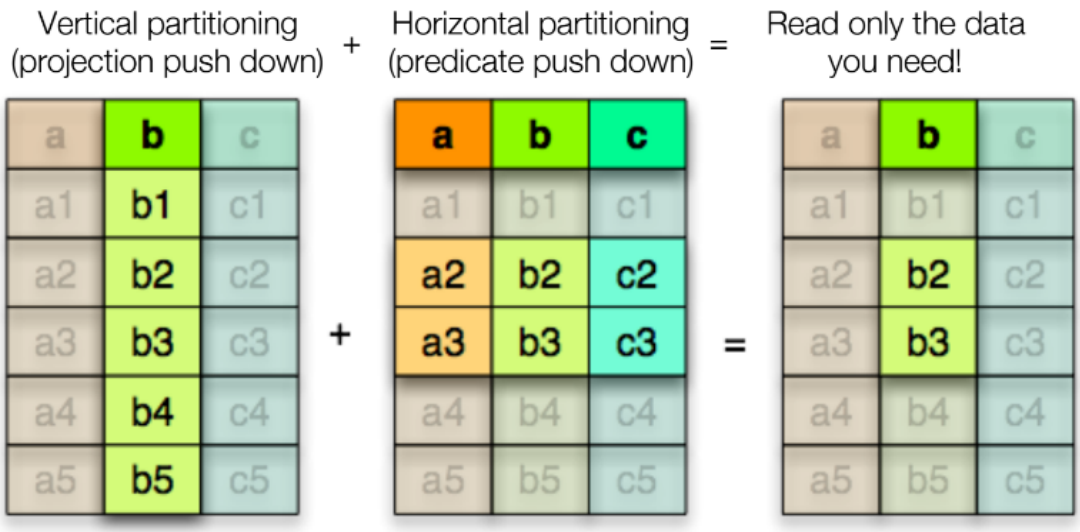

Figure 17: The usage of Predicate Pushdown Filter[22]

– Turn on statistic filter: [23]
In addition to the predicate pushdown filter, applying a statistic filter also

[22]https://www.slideshare.net/cloudera/hadoop-summit-36479635

[23]Parquet performance tuning: The missing guide - Netflix in Strata+Hadoop Conference in New York 2016

supports readers to reduce the amount of time taken for retrieving data from Parquet files. The principle of the statistic filter is simple but highly effective. At the beginning of querying data, a reader starts looking at the metadata of row groups and column trunks. If a row group does not store data the reader needs, then it will skip this row group. The statistic conditions for the filter is minimum and maximum values. If data inside a Parquet file is not ordered, and scattered in different row groups, the statistic filter will not work as reader still reads all row groups to find the data it needs. Then, to apply the statistic filter, the data in Parquet has to be sorted first. For example, there are Parquet files having sorted a column named "Tag" with values from 0 to 15000. These parquet files were split into 3 row groups with 3 segments of "Tag" values [1, 5000], [5001, 10000] and [10001, 15000]. In order to seek all values having "Tag" value in the range from 6000 to 9000, the reader will take a look on metadata of three row groups. As the minimum value the reader looks for is larger than the maximum value of the first row group, the reader can ignore this one. Similarly regarding the third row group [10001, 15000], the maximum value the reader is looking for is smaller than the the minimum value of the third row group, and the reader can skip this row group as well. Finally, the reader only scans and retrieves the data from the second group which contains values it needs, reducing the amount of time to scan and retrieve unnecessary data from the first and third row groups.

4 Experiments

In this section, various experiments have been carried out to validate whether the performance of the system to meet the requirements in several use cases above. The tests aim at increasing the performance of retrieving data from the datastore and executing some calculations, the ability of feeding data from external sources and storing it in Parquet format as well as visualizing raw and aggregated values to the user interface as trends. In the first experiment, several simple computations are conducted with Spark SQL, Impala, and Hive, and the performance of the frameworks are compared with each other. In the second experiment, the data import feature is tested. Data is firstly collected from external sources, processed and stored in a bucket in S3. Finally, the experiment on fetching data from S3 and showing it as a trend to the user.

4.1 Environment Setup

This section will present the platform and the tools to build the prototype. Also, for the performance testing, a data set which has a format similar to data collected from IOT sensors was made up.

4.1.1 Dataset

From the requirements of the use cases, one of the primary goals to achieve was that the prototype could handle data from one million channels or (tags) in a certain amount of time, so a new dataset consisting of a million channels was made up. Also, to manage channels efficiently, channels were divided into multiple groups. As a result, the system included 1000 groups, and each group contained 1000 channels. All of the groups were named from 1 to 1000, and the tags in every group were indicated by a number from 1 to 1000. Splitting data into smaller groups also helped the system not to suffer from poor performance when the reader had to look up the data for only a few channels out of one million channels. Therefore, a data point was presented by time at which it was gathered, its tag and the group which its tag belonged to.

Following the storage design above, information on date, month and year will be separated into different fields, including Year, Month and Date and Time. Both of them have Integer type, and time in a day is defined by the number of seconds. The total number of seconds in a day is 86400. For example, a data point is collected at 13:45:19 on 19th March 2017 is presented as (2017, 3, 19, 3499). Finally, data collected have the same format, so they are formed into a single table. The schema of the table has several fields, including year (INT), month (INT), day (INT), time (INT), group (INT), tag (INT), value (DOUBLE). An example of a table is presented in Table 1.

Table 1: An example of values of tag 6 (group 2) is presented in the table

year	month	day	time	group	tag	value
2017	3	17	3496	2	6	1.3456669
2017	3	17	3497	2	6	-0.9478129
2017	3	17	3498	2	6	1.594354
2017	3	17	93498	2	6	2.0212568

The dataset was made up from a bidding dataset with real numeric bidding data[24]. At first, the capacity of the original dataset was smaller than expected, so data was replicated to be suitable for the requirements of the experiments. Then, as the design of datastore, the dataset is partitioned into different years, months, dates and groups. This means that all the data points having the same year, month and date and belonging to the same group are stored in the same place. Partitioning the data helps the Parquet readers quickly identify where the data is placed. Also when the data is split into smaller trunks, it is selected more precisely to transfer to the computing cluster instead of a huge block of data containing the unnecessary data points. After being divided, all fields are used to be partitioned by becoming names of directories in the datastore and the data is stored in files that only contain time, tag and value. The organization of the dataset in our bucket in S3 is shown in Figure 18. As a result, the final dataset includes 2000 partitions with 6000 files in the Parquet format, and each partition is around 350 MB. The total data set is around 690 GB. This amount of data is equivalent to raw data of two days from a million sensors with the sample rate being one sample per a second. The dataset

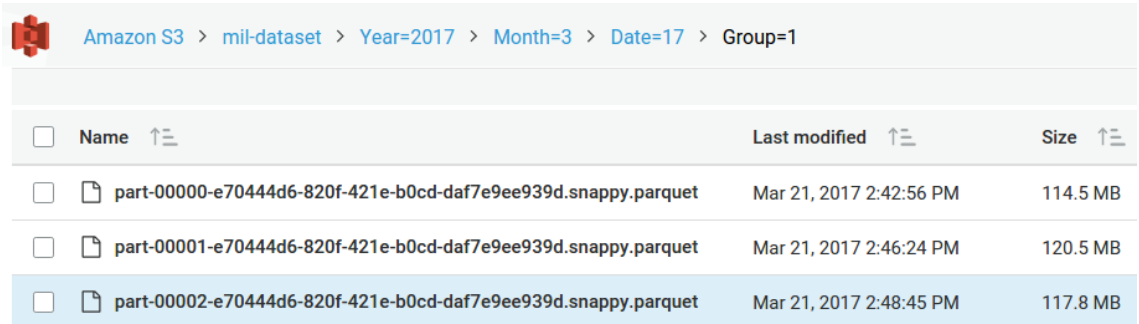

Figure 18: The layout of datastore in S3

was generated by a Spark application to S3 for one group of a day, and then it was replicated for different groups and different days.

4.1.2 AWS

All the experiments were conducted on Amazon Cloud, including the cluster of EC2 instances for computing cluster and S3 for storing the dataset. The type of an EC2

[24]https://www.kaggle.com/zurfer/rtb

instance used for the experiments is R4.2xlarge, and for coding and testing T2.xlarge. R4.2xlarge type was optimized for applications requiring a large capacity of RAM, and was recommended by Amazon. Also, the bandwidth of the network between nodes in a cluster was 10 Gbps, guaranteeing the network latency is minimized. The specification of the two types of EC2 instances is presented in Table 2. Also, one of goals was to examine whether the performance of the system enhances linearly with the number of nodes added to the cluster or not. So each test was performed in clusters with different sizes, including four, eight and sixteen nodes.

Table 2: EC2 Types [25]

	M4.2xlarge	R4.2xlarge
vCPU	4	8
Memory (GiB)	16	61
Network Performance (Gbps)	-	10 Gbps

4.1.3 Hadoop distribution

Regarding installation and deploying the frameworks of the prototype system, it would be slow and not convenient if each of them was set up manually. Therefore, one of three Hadoop distributions, including including Cloudera[26], Hortonworks[27] and MapR[28], was chosen for deploying testing clusters. While MapR distribution is in closed source, both CDH (Cloudera) and HDP (Horton-works) are open source. CDH provides all required components and it has a support from a large community of users, so CDH was chosen as the platform of the testing system. Also, Cloudera provides Cloudera Manager, an application that manages all of the components in CDH. This helps to reduce administrative costs. For these experiments, the testing system was built through Cloudera Manager 5.10 and CHD 5.10. In addition to the required elements of systems and essential components of a Hadoop cluster, several utilities such as Hue were installed for debugging and testing.
With Cloudera Manager, it was easy to add or remove hosts from a cluster and deploy the configuration of a current system to new hosts.

4.2 Experiments

This section explains how the experiments were implemented, and now the settings and optimizations were applied to components of the prototype system to achieve high performance. The results of the tests are presented and discussed in Section 5.

[25]https://aws.amazon.com/ec2/instance-types/
[26]https://www.cloudera.com/products/product-components/cloudera-manager.html
[27]https://hortonworks.com/products/data-center/hdp/
[28]https://mapr.com/products/mapr-distribution-including-apache-hadoop/

4.2.1 Retrieve Data from Datastore

In this experiment, we conduct different SQL queries to retrieve data from the S3 store and do computation. They are count, sum and average queries. The detail of the queries is in Appendix A. The goal of this test is to examine the speed of the system to meet the requirements of retrieving one million data point per second. The experiment was conducted with Spark SQL, Impala and Hive and the performances of these frameworks were compared. This test was carried out to fulfill the requirements of the Use Case 7.

Spark SQL
The Spark application used for testing is run on top of a YARN cluster, and the deployment mode is Client, in which Spark driver is on the same host where jobs were submitted. In contrast to the Client mode, the Spark drive program in Cluster mode is placed on the same host as the Spark Application Master. The benefit of the Cluster Mode is that the client starting the application does not need to keep running during application's lifetime. However, it was difficult to keep track of the progress of Spark application when it was running in Cluster mode, so all of the experiments with SparkSQL was deployed with Client Mode in Spark-shell[29]. As mentioned above, to maximize the performance of a Spark cluster on YARN, we have to utilize the resource YARN provides for a Spark application, including the number of executors, the number of cores in each executor and the amount of RAM supplied for each executor. For instance, one of the configurations used for starting a Spark application to benchmark the performance of cluster of four nodes was:

```
spark2-shell --master yarn \
             --deploy-mode client \
             --executor-cores 3 \
             --driver-memory 5g \
             --executor-memory 10g \
             --conf "spark.executor.instances=12" \
             --conf "spark.dynamicAllocation.enabled=false" \
          --verbose
```

The code of the application is presented in Appendix A. The time for executing queries was measured by a manual script inside the code as well as through Spark web UI.

Impala
Impala, as well as Hive, was deployed on the same nodes in the cluster with SparkSQL, so the number of Impala daemon (Impalad) used for testing was also the same as the SparkSQL node, including four, eight and sixteen nodes. To guarantee the test results were not affected by network latency, an Impala-shell shipped with Impala daemon in one of the nodes was used.

[29]http://spark.apache.org/docs/2.1.0/quick-start.html

Typically, an external table of Impala has to be created to explore data in existing sources through Impala. However, this approach only works with unpartitioned datasets or small partitioned tables. This was occasionally changed as Impala does not support automatic recognition of data in subdirectories recursively under the location provided. To help Impala to identify sub-partitions, users and system administrators have to manually add these partitions to the external table through data manipulation statements. With a huge dataset having a high rate of data generation (1000 partitions per day), this solution may lead to missing data or an inconsistent state of a datastore. In addition, as mentioned above, Impala can synchronize metadata and exploring data in external tables from Hive through its catalog services, and Hive also supports accessing data recursively in sub-partitions of the datastore. Therefore, instead of manually updating new partitions for a datastore, we can create an external table through Hive and Impala and then synchronize the metadata of the table through Hive Metastore. Whenever new partitions are created, the Impala table should be re-updated with new metadata of new partitions from Hive Metastore.

To speed up queries in Impala, the technique "Compute Incremental Stats", mentioned in Section 3.4.2 was applied. However, due to the large volume, a statement computing incremental statistics of a dataset takes 70 minutes to finish.

Hive

Hive based on the MapReduce v2 framework was compared with Spark SQL and Impala on exploring a massive dataset. Similar to Impala, to eliminate the effect of network latency, all experiments with Hive also executed with beeline, a Command Line Interface (CLI) connecting to Hive server 2 through a JDBC/ODBC driver. Then, an instance of Beeline was started in the same node with a Hive server. As mentioned in the previous section, a Hive table should be created for exploring data instead of an Impala table. The statement was used to create external Hive table on an existing dataset on S3:

```
CREATE EXTERNAL TABLE mil_dataset(
time INT,
tag INT,
value DOUBLE)
PARTITIONED BY(
year INT,
month INT,
day INT,
cgroup INT)
STORED AS parquet
LOCATION 's3a://xxxx/mil-dataset/';
```

Also, the configuration[30] should be set to the value "true" to enable Hive to access the partitions of a dataset recursively:

```
set hive.mapred.supports.subdirectories = true
```

[30]https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties

4.2.2 Import Data from external sources to storage in S3

This experiment aims at executing conversion a massive raw data from external sources to a dataset in a Parquet format on S3. A Postgres SQL server was used to simulate external data sources. The schema of the Postgres database is the same as the schema of the dataset on S3, including Year, Month, Day, Time, Group, Tag, and Value. The process of data conversion is shown in Figure 19.

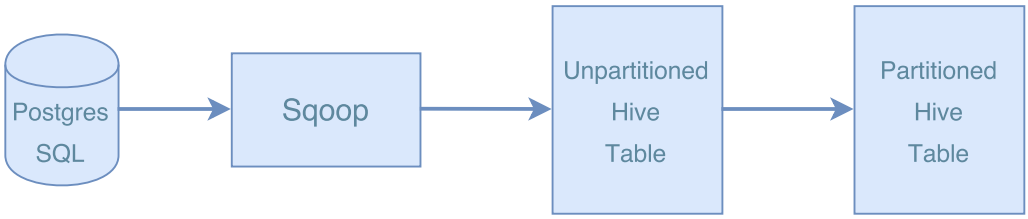

Figure 19: The process of importing data from Postgres SQL to dataset

At first, all the rows in the Postgres table was loaded to Sqoop. Sqoop supports various formats such as text, JSON, Avro for a data storage in the Hadoop ecosystem. One convenient way to convert data to Parquet files is through a Hive table. However, Sqoop is having a limitation in partitioned table, only accepting a column of the String type be partitioned as a key, while all of partitioned keys of the dataset were the Integer type. Therefore, the data was transferred to a temporary unpartitioned Hive Table in a text format. Saving data into any other format would cost the system to spend time in encoding and decoding data. Then, from this temporary table, data was partitioned and converted into Parquet files. At the time this experiment was conducted, Sqoop2 did not support data conversion to a Hive Table yet, so all tasks were executed on Sqoop1. This experiment was carried out to fulfill the requirement of the use case 2.

4.2.3 Data Visualization

The goal of this test focused on retrieving data in Parquet format on datastore and then display it as graphics on the user interface. Data to display includes both raw and aggregated values. In several cases, raw data is required to be aggregated on demand (first aggregate then visualize immediately). However, on this test, data was pre-aggregated and stored in S3 in advance. Raw data and aggregated data have different store locations. Except for the one-second time level (raw data), each remaining time level on the user interface is equivalent to each level of aggregated data, such as ten seconds, one minute, two minutes, ten minutes and one hour.

From the user interface, a small test site was developed, and user could select different days and time levels of data aggregation to display. The small testing site was developed on Bokeh Framework in Python, and it was connected to LIVY through a RESTful service. As mentioned above, Livy provides the user with two ways to interact with Spark in the backend, including the batch mode (through

Spark-submit) and the interactive mode (through Spark-shell). In this test, the interpreter mode was chosen due to its convenience on debugging and changing code. Otherwise, the code was compiled as jar files and submitted to the Spark whenever code was changed. This experiment was conducted for the 3rd, 5th and 6th use cases. This test was carried out to fulfill the requirements of the Use Cases 3, 5 and 6.

5 Results

This section shows the outcomes of the experiments described in Section 4 as well as discussion on the results.

5.1 Retrieve Data from Datastore

The tests are conducted with three different frameworks, including SparkSQL 2.0.0, Impala 2.8.0 and Hive 1.1.0. While Impala and Hive packages were shipped with CDH as default, Spark 2.0.0 installed from an external package. Both tests were implemented in clusters with different sizes, including four, eight and sixteen machines typed R4.2xlarge. An EC2 instance of R4.2xlarge includes an 8-core CPU, 61 GB of RAM and 10 Gbps of network bandwidth. The dataset used for testing was described in Section 4.1.1. Queries and codes were used for these experiments were shown in Appendix A. The results of the tests are presented in Figures 20, 21 and 22.

Based on the results collected from the experiments, it is clear that SparkSQL has

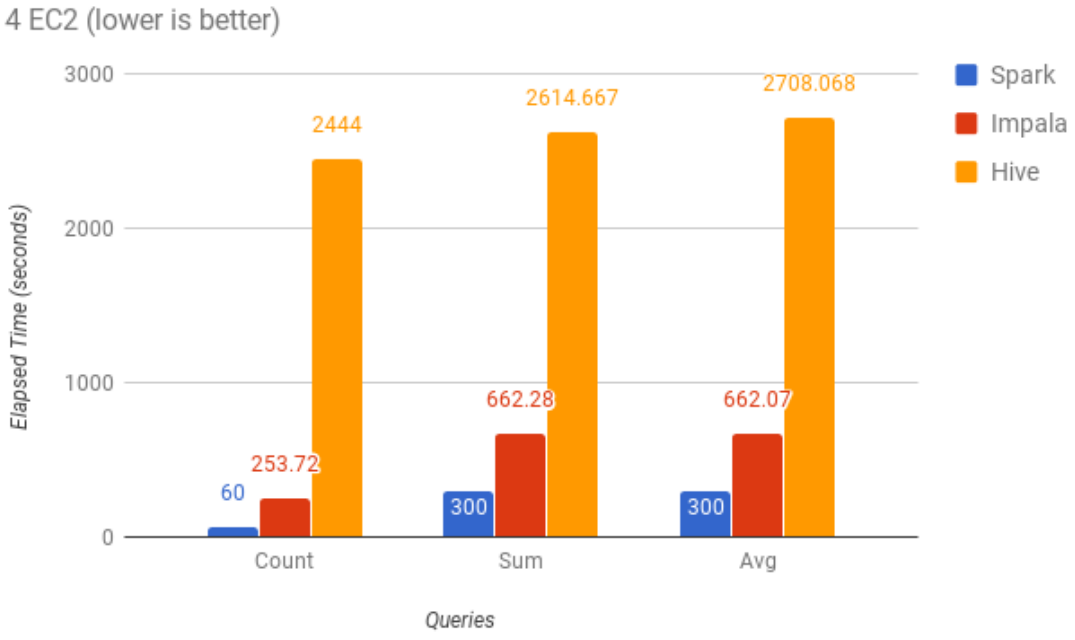

Figure 20: Comparison between Spark, Impala and Hive in a cluster of 4 nodes

better performance than Impala and Hive on both three queries: count, sum, and average. As expected, compared to Hive on MapReduce 2, SparkSQL is outstanding with the performance being better nine or ten times than Hive. Also, the performance of Spark SQL is double than Impala on queries calculating sum and average, and

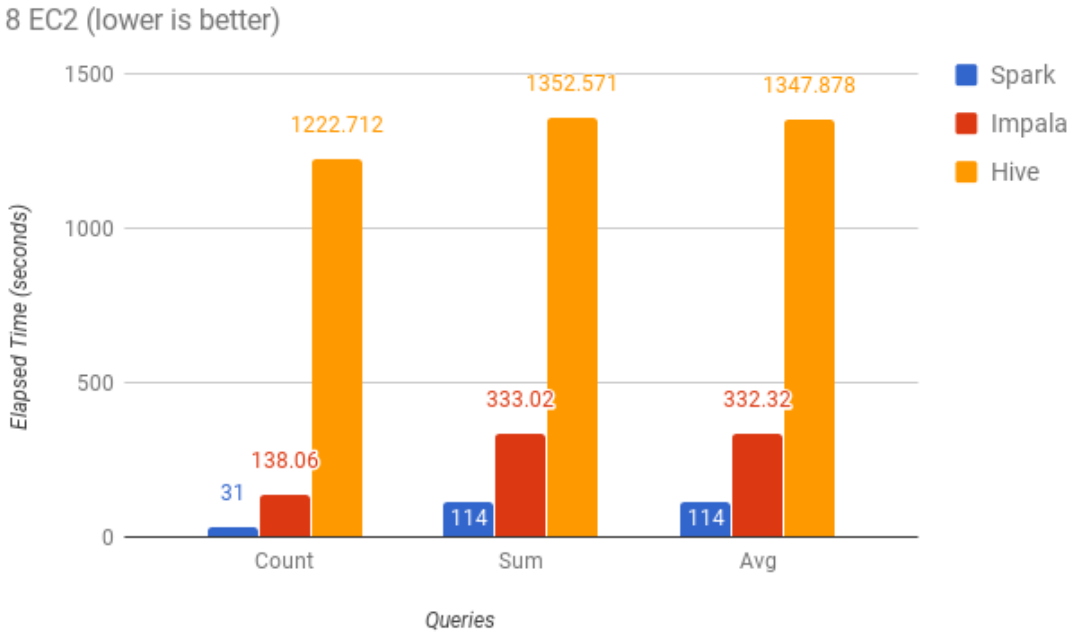

Figure 21: Comparison between Spark, Impala and Hive in a cluster of 8 nodes

almost triple times on counting queries. This can be explained by Spark 2.0 having improvements in performance with the second generation of Project Tungsten. Note that, both of queries ran on a massive number of data points: 172,8 million data points. With a cluster of 16 instances, the speed of the Spark SQL on the query of 'average' reaches 2.4 million data points per second, and the rate of Impala engine also reaches around 1 million data points per second. Also, Spark SQL reaches a speed of 1 million data points per second at a cluster of 8 computers. These rates satisfy the requirements of the use case 7 that reading speed should be one million data points per second.

Also, the performance of a cluster of Impala increased linearly with the number of nodes added to the cluster, and the speedup ratio of the cluster of Spark SQL is higher than the other frameworks, illustrated in Figure 23.

5.2 Import Data from external sources to storage in S3

All of the data stored in PostgresSQL was imported to Parquet files successfully. However, because the data had to be transformed twice, first from Postgres SQL to temporary Hive table and then from the temporary table to partitioned table, the system took such long amount of time to finish the task, even with a small dataset (32 MB). Therefore, in order not to affect the performance of the whole system, importing data in batches can be considered to be run at once or twice a day, when the number of users is low.

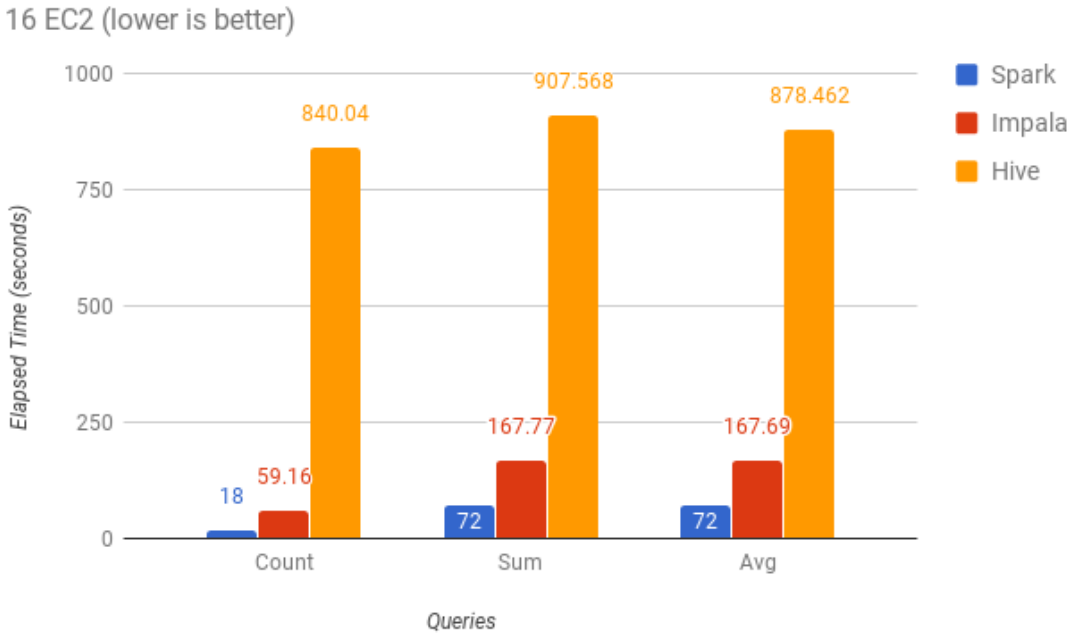

Figure 22: Comparison between Spark, Impala, and Hive in a cluster of 16 nodes

5.3 Visualization of data

In Figures 24, 25 and 26, the user interface is displaying data as a trend, and the user can choose a duration of time (year, month, date) and different time levels from lists. The data of channel one belonging to the group 1 was used for this experiment. In both Figures, data on 17 March 2017 was visualized with time levels including 1 second, 10 minutes and one hour. In Figure 24, 86400 raw data points were displayed, while 144 and 24 aggregated data points were shown in Figures 25 and 26 respectively.

Although the process of data visualization works properly, web user interface takes quite a long time for starting and responding for displaying the plots. This may be because a Spark Context is created and datasets are loaded when a user posts a request. In future work, the response time of the user interface can be reduced by setting up an available Spark Context and loading or caching datasets in this Context.

5.4 Evaluation

The outcomes of tests show that the performance of the prototype might fulfill the performance requirements and features from the use cases. The list of use cases' state is shown in Table 3. The outcome of experiments are discussed as below.

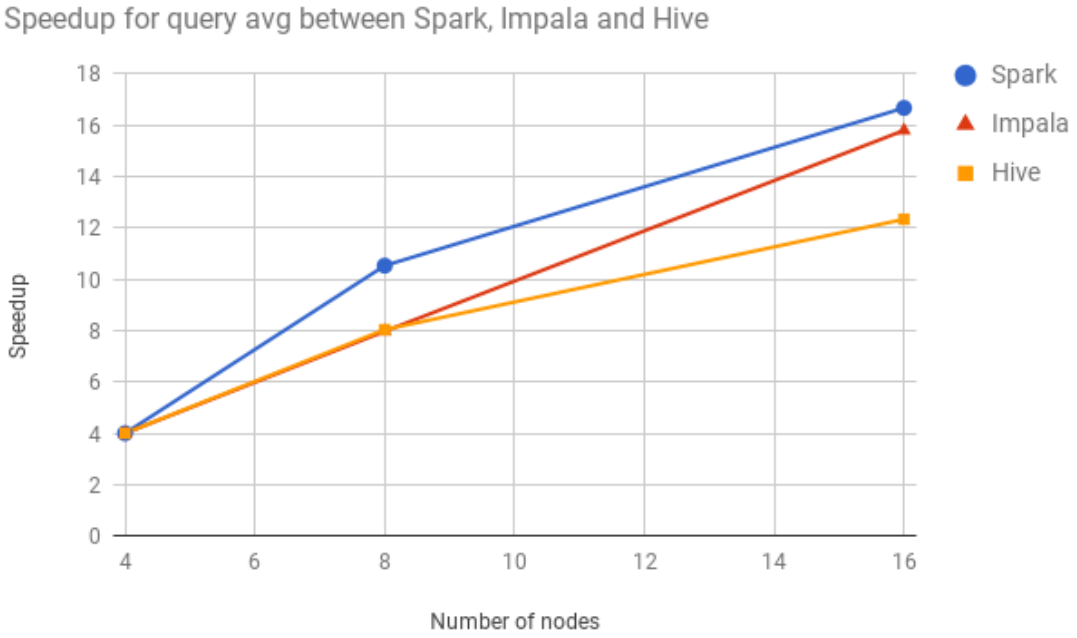

Figure 23: The speedup ratio of Spark, Impala, and Hive with different cluster sizes in query 'avg'

- **Use case 2**

The testing data pipeline may satisfy the demand of feeding raw input data in batches. Due to the limitation of transfer tool, data has been converted to unpartitioned format before saving in a partitioned format as design. Also, saving data into a temporary table causes poor performance on the system. Next investigations should be conducted to reduce wasted resource and run time to storing data in a temporary table, and find a comprehensive solution for the issue.

- **Use case 3**

The requirement of this use case is also fulfilled by the third experiment. After raw input data was loaded in a batch to the datastore, Spark periodically loads and aggregates the new data with different time levels such as ten seconds, ten minutes and one hour, then saving pre-aggregated data into the datastore.

- **Use case 5**

The experiment shows that raw data is load from the datastore and displayed on user graphic interface. In this test, Spark SQL was used for retrieving data from datastore, so there is a delay in time to visualize data to users because the system has to create a new SparkContext at the beginning, and scan data in datastore. In future work, SparkContext may be created in advance and kept in a long time, as well as dataset is cached in SparkSQL, to increase speed data retrieve.

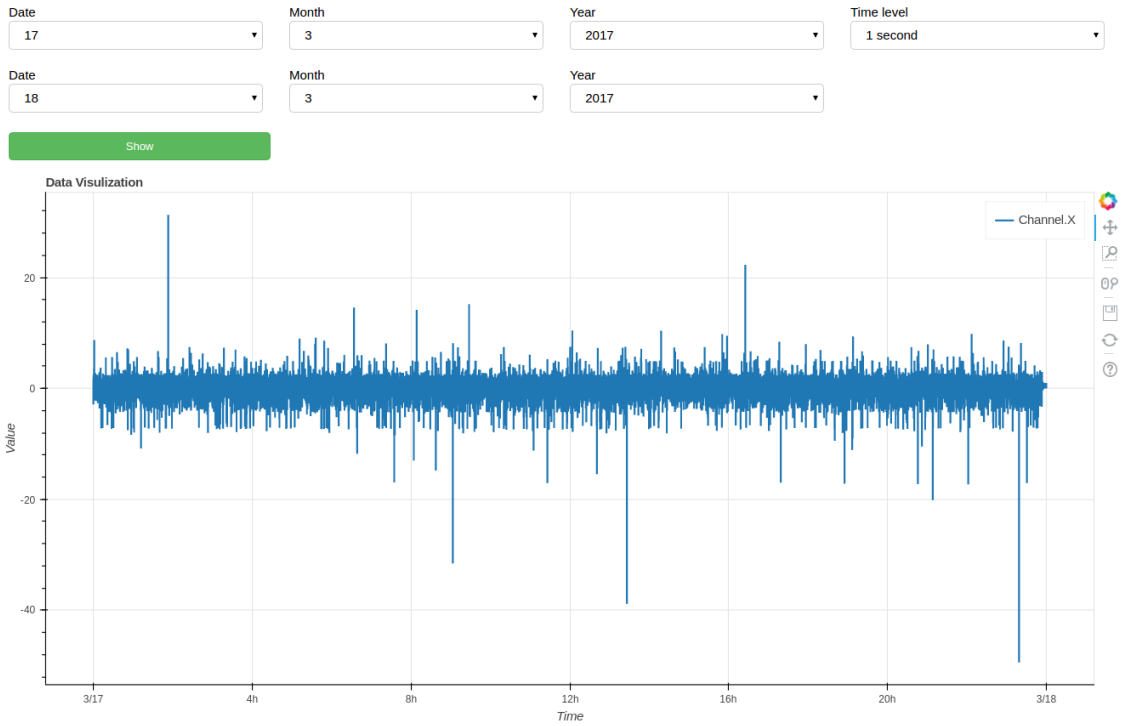

Figure 24: Visualization of raw data with the time level of one second.

- **Use case 6**

Similarly to use case 5, the third experiment also fulfills use case 6. In use case 6, as the number of aggregated data points is smaller than the number of raw data points, a delay in time to visualize data is shorter.

- **Use case 7**

In this use case, the result of the experiment shows that the prototype system can satisfy requirements. Speed data retrieval reaches 2.4 million data points per second with Spark SQL in a cluster of 16 nodes and 1 million data points per second with Impala in a cluster of 16 nodes even data is pulling from S3.

5.5 Cost

This section describes cost to run clusters in AWS in details. Clusters are deployed in EC2 instances, while data is stored in S3. The price of storage in S3 is Table 4. Regarding EC2 instances, three types of EC2 can be used for building a cluster to analyze data like the previous experiments, including M4.2xlarge, R4.2xlarge, and C4.4xlarge. While type M4.2xlarge may meet the demand of regular workload and calculations, type R4.2xlarge with a large amount of RAM can satisfy requirement of calculations on a huge dataset, and C4.4xlarge may be suitable for extensive and complex calculations. The specification and price of these types are shown in Table 5. Note that cost of EC2 instances and S3 storage were calculated based on region

Table 3: State of use cases

Use case	Description	Experiments
1	Feeding raw input data in real time	No
2	Feeding raw input data in batches	Yes
3	Aggregating raw input data into averaged data	Yes
4	Dashboards	No
5	Displaying raw data in a trend	Yes
6	Displaying averaged data in a trend	Yes
7	Calculating the results from a single column of data	Yes
8	Calculating the results between pairs of columns of data	No
9	Search a single column of raw data for a time range based on the value of data in a piecewise constant measurement	No
10	Finding measurements belonging to a category (grade, lab, dead)	No
11	Finding control loops that fail validation	No
12	Finding measurements with high correlation with the current measurement among all the measurements in the system	No
13	Finding measurements with similar spectral peaks as the current measurement among all the measurements in the system	No

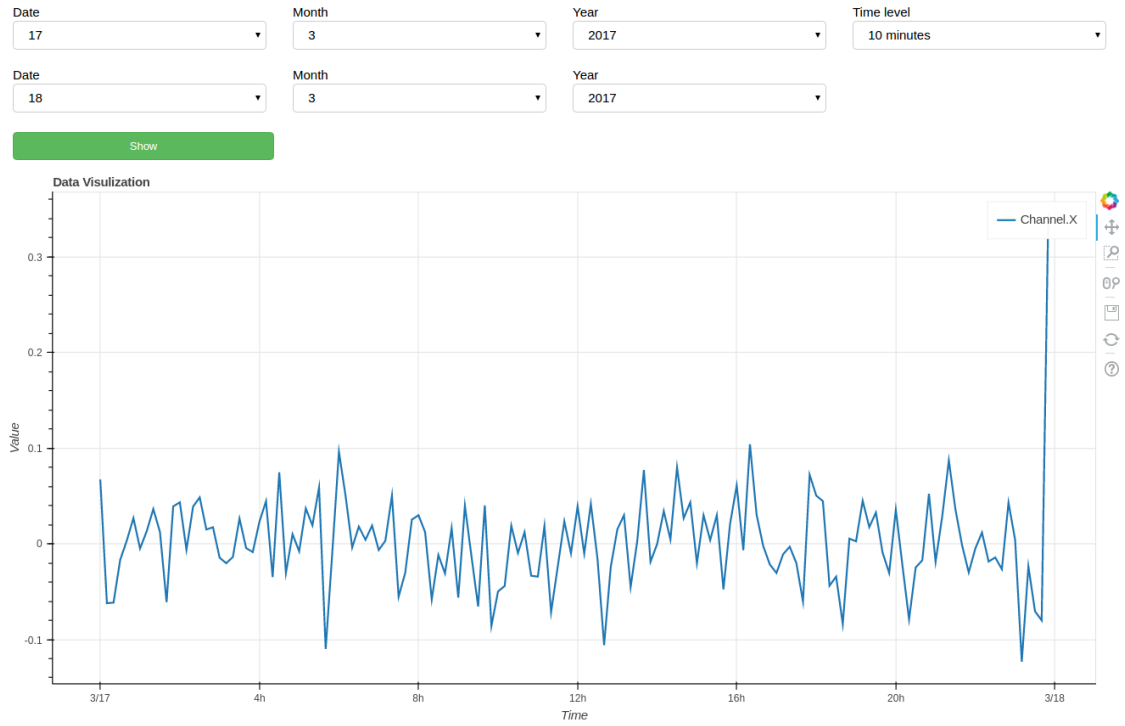

Figure 25: Visualization of aggregated data with the time level of ten minutes.

eu-central-1 (Frankfurt).

Table 4: S3 storage pricing [31]

	Price
First 50 TB/ month	$0.0245 per GB
Next 450 TB/ month	$0.0235 per GB
Over 500 TB/ month	$0.0225 per GB

Table 5: THe specification and price of different EC2 types [32]

	M4.2xlarge	R4.2xlarge	C4.4xlarge
vCPU	8	8	16
Memory (GiB)	32	61	30
Network (Gbps)	1	10	2
Prices/ hour	$0.48	$0.64	$0.909

For example, based on experiments above and the cost to run a cluster for doing the tests on AWS (details on the Table 2, the monthly cost of platform for a big dataset

[31] https://aws.amazon.com/s3/pricing/
[32] https://aws.amazon.com/ec2/pricing/on-demand/

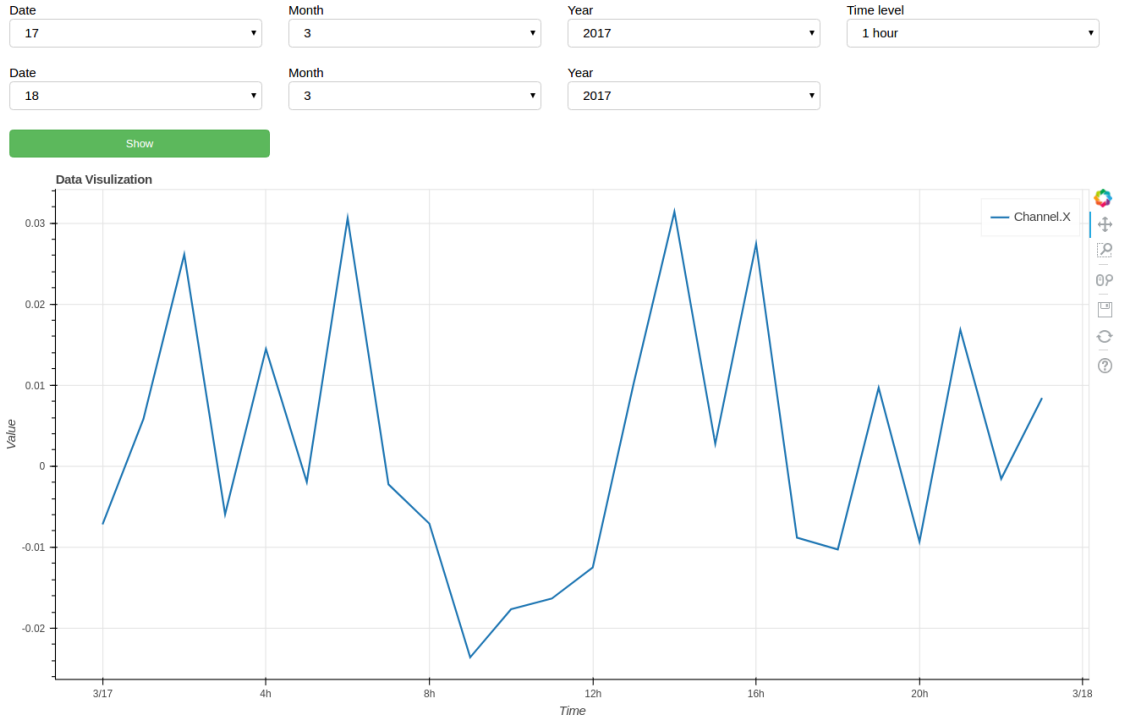

Figure 26: Visualization of aggregated data with the time level of one hour.

should be $5009.85 per month, including $3686.4 for a cluster of 8 nodes and $1254.4 for 50 TB data on S3. The configuration of the system includes:

- The cluster of 8 instances of EC2 R4.2xlarge
- 50 TB data on S3
- 1 million channels
- 10,000,000 PUT, COPY, POST, or LIST requests
- 35,000,000 GET and all other requests

Table 6: Monthly cost for running a cluster on AWS

	Description	Price
EC2	Type: R4.2xlarge	$0.64/ machine / hour
S3	Storage: First 50 TB of storage	$0.0245 / GB / month
	Request: PUT, COPY, POST, or LIST	$0.0054 / 1,000 requests
	Request: GET and all other ones	$0.0043 / 10,000 requests

6 Conclusion

When Industry 4.0 emerges, the number of sensors and machines connected to the Internet raises significantly. As a result, log data generated from these devices also witnesses an exponential growth in volume and enormously complex calculations are needed for gaining valuable insight are required to apply to this data. Therefore, various frameworks and software such as Spark, Hive, Cassandra, and HBase have been developed to store and explore this massive amount of data. The thesis is aimed at evaluating how effectively the current Big Data Frameworks and Tools manipulate industrial Big Data, especially processing data.

6.1 Achievements

The thesis achieved the initial goal, which is to build a prototype of a data pipeline handling a massive amount of industrial data. The prototype contains several components, which are:

- Spark: is used for pre-aggregating raw input data periodically, retrieving data and executing complex calculations on data.
- Sqoop: is used for transfer and convert data from external sources to files in Parquet format, stored in S3.
- Impala: is used for retrieving data without a delay and performance uncomplicated aggregations.
- Parquet: is the columnar file format used in the datastore. Parquet provides high-speed data retrieval, efficient compress and encode ratios.
- Livy: is a RESTful interface, transferring data exchanged between frontend and Spark.
- Bokeh: is used for visualizing data in user graphic interface.

Several experiments were conducted with the prototype system and the results gained are positive. The prototype satisfied the requirements in use cases 2, 3, 5, 6 and especially, in use case 7, the speed data retrieval reaches 2.4 million data points and double required speed.

6.2 Future Work

Although the prototype has shown that it can meet the requirements of the use cases, there is still some room for enhancement of the performance of the prototype. For example, the amount of RAM in nodes in a cluster is high, but through running log data from big data engines, it seems that these engines did not leverage all the capability of RAM in nodes. A detailed investigation should be conducted to figure out how the resource of cluster utilize efficiently.

Moreover, these experiments should be carried out on different datasets to compare the performance among them. Also, the system should be tested in various scenarios such as load tests and stress tests.

As mentioned in Section 5.4, there are still drawbacks in the prototype should be eliminated. For example, in use case 2, data imported in batch mode was saved in temporary table before being stored in partitioned Parquet files in S3. In use case 5, SparkContext should be initialized in advance, and DataFrame or Dataset of data should be cached in memory to enhance the speed data retrieval and visualization.

The remaining module of system also should be developed and tested to fulfill requirements of other use cases, particularly use cases 12 and 13, which required a huge number calculations on a massive volume of data.

References

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

[3] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. A formal presentation of MongoDB (extended version). *arXiv preprint arXiv:1603.09291*, 2016.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink$^{\text{TM}}$: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43, 1998.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[9] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12):1295–1306, 2014.

[10] John Gantz and David Reinsel. The digital universe in 2020: Big Data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[11] Goetz Graefe. Volcano - An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[12] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011.

[14] Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of Cypher, Gremlin and native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[16] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[18] Jay Lee, Hung-An Kao, and Shanhu Yang. Service innovation and smart analytics for Industry 4.0 and Big Data environment. *Procedia Cirp*, 16:3–8, 2014.

[19] Andrew McAfee, Erik Brynjolfsson, et al. Big Data: The management revolution. *Harvard business review*, 90(10):60–68, 2012.

[20] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of Web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[21] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[23] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 293–307, 2015.

[24] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, 2013.

[25] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. NoSQL databases. *Lecture Notes, Stuttgart Media University*, 2011.

[26] Ronald C Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(12):S1, 2010.

[27] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[28] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16, 2013.

[29] Skye Wanderman-Milne and Nong Li. Runtime code generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

A Code and SQL queries for experiments

A.1 Code for SparkSQL

```
import spark.sqlContext.implicits._

val accessKey     = "xxxx"
val secretKey     = "xxxx"
val endPoint      = "s3.eu-central-1.amazonaws.com"

val sc  = spark.sparkContext
System.setProperty("com.amazonaws.services.s3.enableV4", "true")
sc.hadoopConfiguration.set("fs.s3a.impl",
"org.apache.hadoop.fs.s3a.S3AFileSystem")
sc.hadoopConfiguration.set("fs.s3a.access.key", accessKey)
sc.hadoopConfiguration.set("fs.s3a.secret.key", secretKey)
sc.hadoopConfiguration.set("fs.s3a.endpoint", endPoint)

val link ="mil-dataset"
val df = spark.read.parquet(s"s3a://$link")
import spark.sqlContext.sql

df.filter("Tag = 1").count()
df.filter("Tag = 2").agg(sum($"Value")).show()
df.filter("Tag = 3").agg(mean($"Value")).show()
```

A.2 SQL Queries for Impala and Hive

```
- Count: select count(*) where tag = 1;
- Sum: select sum(value) where tag = 2;
- Average: select avg(value) where tag = 3;
```